# Microsoft® QuickBASIC 4

# Programmieren in BASIC: Ausgewählte Themen

Microsoft®

# Microsoft® QuickBASIC

## Programmieren in BASIC: Ausgewählte Themen

Für IBM® Personal Computer
und kompatible Computer

Microsoft Corporation

Die in diesem Handbuch enthaltenen Angaben sind ohne Gewähr und können ohne weitere Benachrichtigung geändert werden. Die Microsoft Corporation geht hiermit keinerlei Verpflichtungen ein. Die in diesem Handbuch beschriebene Software wird auf Basis eines Lizenzvertrages und einer Verschwiegenheitsverpflichtung (die Verpflichtung die Software nicht weiterzugeben) geliefert. Der Käufer darf eine einzelne Kopie der Software zu Sicherungszwecken (Backup) anfertigen.

Teile dieses Handbuches dürfen weder auf elektronischer noch mechanischer Weise, einschließlich Fotokopien und sonstige Aufzeichnungen, ohne die schriftliche Genehmigung der Microsoft Corporation vervielfältigt oder übertragen werden.

© Copyright 1984-1988 Microsoft Corporation. Alle Rechte vorbehalten.

Microsoft®, MS®, MS-DOS® und CodeView™ sind eingetragene Warenzeichen der Microsoft Corporation.

IBM® ist ein eingetragenes Warenzeichen der International Business Machines Corporation.

WordStar® ist ein eingetragenes Warenzeichen der MicroPro International Corporation.

Artikelnr. 01647
Dokumentnr. 8623-400-00-2

# Inhaltsverzeichnis


**Einleitung**

Über diese BASIC-Version xi

Zusätzliche Quellen zum Erlernen von BASIC und DOS xi

Die Gliederung der Kapitel xii

Typographische Konventionen xiii

Der in diesem Handbuch verwendete Programmierstil xv

**1 Strukturen zur Ablaufsteuerung**

1.1 Wie Sie die Reihenfolge der Ausführung von Anweisungen ändern 1.2

1.2 Boolesche Ausdrücke 1.3

1.3 Entscheidungsstrukturen 1.6

1.3.1 Block-**IF...THEN...ELSE** 1.9

1.3.2 **SELECT CASE** 1.11

1.4 Schleifenstrukturen 1.19

1.4.1 **FOR...NEXT**-Schleifen 1.19

1.4.2 **WHILE...WEND**-Schleifen 1.25

1.4.3 **DO...LOOP**-Schleifen 1.27

1.5 Anwendungsbeispiele 1.33

1.5.1 Programm zur Scheckheftbilanz (SCHECK.BAS) 1.34

1.5.2 Filter für Wagenrücklauf-Zeilenvorschub (WRZV.BAS) 1.35

**2 Prozeduren: Unterprogramme und Funktionen**

2.1 Prozeduren: Bausteine für die Programmierung 2.3

2.2 Prozeduren im Vergleich mit Unterroutinen und Funktionen 2.4

2.2.1 **SUB** und **GOSUB** im Vergleich 2.5

2.2.2 **FUNCTION** im Vergleich mit **DEF FN** 2.74

2.3 Wie Sie Prozeduren definieren 2.9

2.4 Wie Sie Prozeduren aufrufen 2.11

2.4.1 Wie Sie eine **FUNCTION**-Prozedur aufrufen 2.11

2.4.2 Wie Sie eine **SUB**-Prozedur aufrufen 2.12

2.5 Wie Sie Argumente an Prozeduren übergeben 2.14
2.5.1 Parameter und Argumente 2.14
2.5.2 Wie Sie Konstanten und Ausdrücke übergeben 2.16
2.5.3 Wie Sie Variablen übergeben 2.17
2.5.4 Überprüfen von Argumenten mit der Anweisung **DECLARE** 2.22
2.5.5 Wie Sie Argumente nach Referenz übergeben 2.29
2.5.6 Wie Sie Argumente nach Wert übergeben 2.30
2.6 Die gemeinsame Nutzung von Variablen mit **SHARED** 2.32
2.6.1 Die gemeinsame Nutzung von Variablen mit bestimmten Prozeduren in einem Modul 2.32
2.6.2 Die gemeinsame Nutzung von Variablen mit allen Prozeduren in einem Modul 2.34
2.6.3 Die gemeinsame Nutzung von Variablen mit anderen Modulen 2.37
2.6.4 Das Problem von adreßgleichen Variablen 2.39
2.7 Automatische und **STATIC**-Variablen 2.40
2.8 Beibehaltung der Werte von lokalen Variablen mit der **STATIC**-Anweisung 2.41
2.9 Rekursive Prozeduren 2.42
2.9.1 Die Funktion Fakultät 2.43
2.9.2 Anpassen der Stapelgröße 2.44
2.10 Übertragen der Kontrolle an ein anderes Programm mit **CHAIN** 2.45
2.11 Anwendungsbeispiel: Rekursive Verzeichnisdurchsuchung (WOIST.BAS) 2.47
2.12 Zusammenfassung der Anweisungen, die in BASIC-Prozeduren verwendet werden 2.52

**3 Datei- und Geräte-E/A**
3.1 Wie Sie Text auf den Bildschirm ausgeben 3.3
3.1.1 Bildschirmzeilen und -spalten 3.4
3.1.2 Wie Sie Text und Zahlen mit **PRINT** anzeigen 3.5
3.1.3 Wie Sie formatierte Ausgaben mit **PRINT USING** anzeigen 3.6
3.1.4 Überspringen von Leerzeichen und Springen in eine angegebene Spalte 3.7
3.1.5 Wie Sie die Zeilen- und Spaltenzahl verändern 3.8
3.1.6 Wie Sie ein Text-Darstellungsfeld erstellen 3.8
3.2 Wie Sie Eingaben von der Tastatur einlesen 3.10
3.2.1 Die **INPUT**-Anweisung 3.11
3.2.2 Die **LINE INPUT**-Anweisung 3.12
3.2.3 Die **INPUT$**-Funktion 3.14
3.2.4 Die **INKEY$**-Funktion 3.14
3.3 Wie Sie den Text-Cursor steuern 3.15
3.3.1 Wie Sie den Cursor positionieren 3.16
3.3.2 Wie Sie die Cursor-Größe verändern 3.17
3.3.3 Wie Sie Informationen über die Position des Cursors erhalten 3.18

3.4 Wie Sie mit Datendateien arbeiten 3.19
3.4.1 Wie Datendateien organisiert sind 3.19
3.4.2 Sequentielle und Direktzugriffsdateien 3.20
3.4.3 Wie Sie eine Datendatei öffnen 3.20
3.4.4 Wie Sie Datendateien schließen 3.24
3.4.5 Wie Sie sequentielle Dateien verwenden 3.24
3.4.6 Wie Sie Direktzugriffsdateien benutzen 3.33
3.4.7 Binärdatei-E/A 3.43
3.5 Wie Sie mit Geräten arbeiten 3.47
3.5.1 Unterschiede zwischen Geräte- und Datei-E/A 3.48
3.5.2 Datenübertragung über einen seriellen Anschluß 3.50
3.6 Anwendungsbeispiele 3.51
3.6.1 Immerwährender Kalender (KAL.BAS) 3.51
3.6.2 Wie Sie eine Direktzugriffsdatei indizieren (INDEX.BAS) 3.56
3.6.3 Terminal-Emulator (TERMINAL.BAS) 3.64
3.7 Zusammenfassung der Standard-E/A-Anweisungen 3.67
3.8 Zusammenfassung der Datei-E/A-Anweisungen 3.68

**4 Zeichenkettenverarbeitung**
4.1 Wie Sie Zeichenketten definieren 4.2
4.2 Zeichenketten variabler und fester Länge 4.4
4.2.1 Zeichenketten variabler Länge 4.4
4.2.2 Zeichenketten fester Länge 4.5
4.3 Wie Sie Zeichenketten zusammensetzen 4.6
4.4 Wie Sie Zeichenketten vergleichen 4.7
4.5 Wie Sie nach Zeichenketten suchen 4.8
4.6 Wie Sie Teile von Zeichenketten zurückbekommen 4.10
4.6.1 Wie Sie Zeichen von der linken Seite einer Zeichenkette zurückbekommen 4.10
4.6.2 Wie Sie Zeichen von der rechten Seite einer Zeichenkette zurückbekommen 4.12
4.6.3 Wie Sie Zeichen von beliebiger Stelle in einer Zeichenkette zurückbekommen 4.12
4.7 Wie Sie Zeichenketten erzeugen 4.13
4.8 Wie Sie Groß- bzw. Kleinschreibung verändern 4.14
4.9 Zeichenketten und Zahlen 4.15
4.10 Wie Sie Zeichenketten verändern 4.16
4.11 Anwendungsbeispiel: Wie Sie eine Zeichenkette in eine Zahl umwandeln (ZEIINZAH.BAS) 4.17
4.12 Zusammenfassung der Anweisungen und Funktionen zur Zeichenkettenverarbeitung 4.18

**5 Grafiken**

5.1 Was Sie für grafische Programme benötigen 5.3
5.2 Bildpunkte und Bildschirmkoordinaten 5.5
5.3 Das Zeichnen von Grundformen: Punkte, Geraden, Rechtecke 5.7
5.3.1 Wie Sie Punkte mit **PSET** und **PRESET** grafisch darstellen 5.7
5.3.2 Wie Sie mit **LINE** Geraden und Rechtecke zeichnen 5.8
5.4 Wie Sie Kreise und Ellipsen mit **CIRCLE** zeichnen 5.14
5.4.1 Wie Sie Kreise zeichnen 5.14
5.4.2 Wie Sie Ellipsen zeichnen 5.15
5.4.3 Wie Sie Bögen zeichnen 5.17
5.4.4 Wie Sie Tortendiagramme zeichnen 5.21
5.4.5 Wie Sie Formen zeichnen, die mit dem Bildverhältnis anzupassen sind 5.22
5.5 Wie Sie ein grafisches Darstellungsfeld definieren 5.24
5.6 Wie Sie die Koordinaten eines Darstellungsfeldes mit **WINDOW** neu definieren 5.28
5.6.1 Die Reihenfolge von Koordinatenpaaren 5.32
5.6.2 Wie Sie sich logische und physikalische Koordinaten merken 5.33
5.7 Wie Sie Farben verwenden 5.34
5.7.1 Wie Sie eine Farbe für grafische Ausgabe wählen 5.35
5.7.2 Wie Sie die Vordergrund- oder Hintergrundfarbe verändern 5.36
5.7.3 Wie Sie Farben mit **PALETTE** und **PALETTE USING** verändern 5.39
5.8 Wie Sie Formen ausmalen 5.41
5.8.1 Ausmalen mit Farben 5.42
5.8.2 Zeichnen mit Mustern: Ausfüllen mit Mustern 5.44
5.9 **DRAW**: Eine graphische Makro-Sprache 5.55
5.10 Grundlegende Techniken der Animation 5.58
5.10.1 Wie Sie Bilder mit **GET** speichern 5.58
5.10.2 Wie Sie Bilder mit **PUT** bewegen 5.61
5.10.3 Animation mit **GET** und **PUT** 5.65
5.10.4 Animation mit Bildschirmseiten 5.71
5.11 Anwendungsbeispiele 5.73
5.11.1 Balkendiagramm-Generator (BALKEN.BAS) 5.73
5.11.2 Mathematisch erzeugte Farben in einer Figur (MANDEL.BAS) 5.79
5.11.3 Editor für Muster (EDMUST.BAS) 5.86
5.12 Zusammenfassung der Grafikanweisungen und -funktionen 5.92

**6 Fehler- und Ereignisverfolgung**

6.1 Fehlerverfolgung 6.2
- 6.1.1 Wie Sie Fehlerverfolgung aktivieren 6.3
- 6.1.2 Wie Sie eine Fehlerbehandlungsroutine schreiben 6.4

6.2 Ereignisverfolgung 6.9
- 6.2.1 Wie Sie Ereignisse erfassen: Registrierung und Verfolgung 6.9
- 6.2.2 Wie Sie das zu verfolgende Ereignis angeben und Ereignisverfolgung aktivieren 6.10
- 6.2.3 Ereignisse, die BASIC verfolgen kann 6.11
- 6.2.4 Wie Sie Ereignisverfolgung aussetzen oder ausschalten 6.12
- 6.2.5 Wie Sie Tastenbetätigungen verfolgen 6.13
- 6.2.6 Wie Sie Musikereignisse verfolgen 6.18

6.3 Fehler- und Ereignisverfolgung in **SUB**- oder **FUNCTION**-Prozeduren 6.21

6.4 Verfolgung in mehrmoduligen Programmen 6.22
- 6.4.1 Ereignisverfolgung quer durch Module 6.22
- 6.4.2 Fehlerverfolgung in mehrmoduligen Programmen 6.23

6.5 Fehler- und Ereignisverfolgung in Programmen, die mit bc kompiliert sind 6.26

6.6 Anwendungsbeispiel: Verfolgen von Dateizugriffsfehlern (DATFEHL.BAS) 6.28

6.7 Zusammenfassung der Anweisungen und Funktionen zur Verfolgung 6.31

# Einleitung

Seitdem die Microsoft Corporation 1975 das erste BASIC für Microcomputer vorgestellt hat, haben Anwender um weitere Informationen zur Benutzung dieser populären Programmiersprache gebeten. Dieses Handbuch wurde geschrieben, um diesen Wunsch zu erfüllen. *Programmieren in BASIC: Ausgewählte Themen* erörtert diejenigen Themen, nach denen Anwender am häufigsten gefragt haben, und zeigt Ihnen gleichzeitig, wie Sie neue leistungsfähige Spracheigenschaften wie benutzerdefinierte Datentypen, erweiterte Schleifen- und Entscheidungsstrukturen sowie Prozeduren einsetzen. Dieses Handbuch konzentriert sich auf Lösungen häufig vorkommender Programmierprobleme mit BASIC und enthält vollständige Programme, die Ihnen als Beispiele dienen sollen.

Das Handbuch gliedert sich wie folgt:

- Kapitel 1,"Strukturen zur Ablaufsteuerung"

  zeigt Ihnen, wie Sie Schleifen- und Entscheidungsstrukturen verwenden, um den Ablauf Ihrer Programmsteuerung zu verändern. Wenn Sie BASIC seit einiger Zeit nicht mehr benutzt haben, werden Sie von der zusätzlichen Vielseitigkeit, die die folgenden neuen Anweisungen zur Ablaufsteuerung bieten, angenehm überrascht sein:

  - **DO...LOOP**-Schleifen
  - Block-**IF...THEN...ELSE**-Anweisungen
  - **SELECT CASE**-Anweisungen

- Kapitel 2, "Prozeduren: Unterprogramme und Funktionen"

  zeigt Ihnen, wie **SUB**- und **FUNCTION**-Prozeduren geschrieben werden. Prozeduren ermöglichen es Ihnen, unabhängige Routinen zu erstellen, die Sie dann wie BASIC-Anweisungen und -Funktionen einsetzen können.

- Kapitel 3, "Datei- und Geräte-E/A"

  zeigt Ihnen, wie Sie BASIC-Programme schreiben, um auf Diskettendateien zuzugreifen und mit Geräten zu kommunizieren. Lesen Sie dieses Kapitel um herauszufinden, wie Sie mit den Anweisungen **TYPE...END TYPE** und **DIM** einen Verbund definieren, und dann diesen Verbund (Record) für Direktzugriffsdatei-E/A benutzen können.

- Kapitel 4, "Verarbeitung von Zeichenketten"

  erläutert, wie mit BASIC Zeichensequenzen manipuliert werden können. Schlagen Sie in diesem Kapitel nach, um mehr zu finden über:

  – Zeichenketten fester Länge
  – Neue Funktionen, die Leerzeichen entweder vom Ende oder vom Beginn einer Zeichenkette entfernen oder Großbuchstaben in Kleinbuchstaben umwandeln (und umgekehrt)

- Kapitel 5, "Grafiken"

  zeigt Ihnen, wie Sie BASIC verwenden, um Figuren auf dem Bildschirm zu zeichnen, anschließend deren Farbe zu verändern, sie an verschiedene Stellen des Bildschirms zu bewegen, sie mit Farben auszumalen, sie mit Mustern zu versehen, oder sie zu drehen.

- Kapitel 6, "Fehler- und Ereignisverfolgung"

  zeigt Ihnen, wie Sie Programme schreiben, die Laufzeitfehler und -ereignisse verfolgen und behandeln.

Die beiden anderen Handbücher aus dieser Reihe enthalten folgende Informationen:

| ***Handbuch*** | ***Information*** |
|---|---|
| *Lernen und Anwenden von Microsoft QuickBASIC* | Eine umfassende Anleitung zur Benutzung der Microsoft QuickBASIC-Programmentwicklungsumgebung: Editieren, Debuggen, QuickBASIC-Menüs und -Dialogfelder, Erstellen von Quick-Bibliotheken, Kompilieren und Binden aus DOS heraus.<br>Die Anhänge dieses Handbuchs zeigen Ihnen, wie Sie in früheren BASIC-Versionen geschriebene Programme konvertieren können, und wie Sie Programme, die in C und Assembler geschrieben sind, mit BASIC verwenden können.<br>Das Glossar in diesem Handbuch erklärt neue und unbekannte Begriffe, die in allen drei Handbüchern verwendet werden. |
| *BASIC-Befehls-verzeichnis* | Ein alphabetisches Nachschlagewerk für die mehr als 200 Anweisungen und Funktionen von BASIC.<br>Dieses Handbuch beschreibt ebenfalls die Elemente von BASIC: Datentypen, Variablen, Konstanten, Ausdrücke und Operatoren.<br>Die Anhänge dieses Handbuches umfassen eine ASCII-Tabelle, eine Liste der reservierten BASIC-Worte und eine vollständige Liste aller Fehlermeldungen, auf die Sie bei der Programmierung in BASIC treffen könnten. |

# Über diese BASIC-Version

Wenn Sie bisher mit einem interpretierten BASIC wie BASICA oder GW-BASIC programmiert haben, werden Sie feststellen, daß dieses BASIC neue Anweisungen, Funktionen und Eigenschaften hinzufügt, die Ihr Programm strukturierter, schneller und leichter wartbar machen. Dies alles wird bei gleichzeitiger Beibehaltung der Einfachheit einer Ihnen vertrauten Programmentwicklung erreicht. Die Programmgeschwindigkeit wird jedoch nicht einer leichten Anwendung geopfert. Laden Sie eines Ihrer alten BASIC-Programme in QuickBASIC, kompilieren Sie es, und starten Sie es dann. Abhänging vom Typ des Programmes, das Sie gewählt haben, werden Sie feststellen, daß Ihr Programm 30 bis 40 mal schneller ist als mit BASICA.

Mit dieser neuen BASIC-Version wird BASIC zu einem echten Entwicklungswerkzeug für den ernsthaften Programmierer. Gleichzeitig bleibt es die ideale Sprache für Programmieranfänger und Gelegenheitsprogrammierer, die nicht notwendigerweise professionelle Programmierer sind, aber eine Programmiersprache benötigen, die Ihnen dabei hilft, Probleme schnell und effizient zu lösen.

***Hinweis*** Der Begriff "BASICA" wird in diesem Handbuch stellvertretend für interpretiertes BASIC verwendet.

## Zusätzliche Quellen zum Erlernen von BASIC und DOS

Dieses Handbuch geht davon aus, daß Sie bereits einigermaßen mit fundamentalen Programmierkonzepten, wie z. B. Variablen, vertraut sind. Zudem wird angenommen, daß Sie die Grundlagen im Umgang mit DOS kennen, wie die Anwendung der Befehle **COPY** und **DIR**.

### BASIC

Wenn Sie zum ersten Mal mit BASIC arbeiten, oder Sie zusätzliche Informationen zur Programmierung in BASIC benötigen, können die folgenden Bücher für Sie hilfreich sein:

Dwyer, Thomas A., und Margot Critchfield. *BASIC and the Personal Computer.* Reading, Mass.: Addison-Wesley Publishing Co., 1978.

Enders, Bernd, und Bob Petersen. *BASIC Primer for the IBM PC & XT.* New York, NY.: New American Library, 1984.

Hergert, Douglas. *Microsoft QuickBASIC.* 2. Ausgabe. Redmond, Wash.: Microsoft Press. In Vorbereitung.

Die erste Ausgabe dieses Buches erläutert Programmiertechniken für die QuickBASIC-Versionen 2.0 und 3.0. Verwenden Sie die weitergehende zweite Ausgabe für Informationen zur Programmierung mit QuickBASIC Version 4.0.

### DOS

Wenn Sie weitere Informationen zu DOS benötigen, versuchen Sie es mit folgenden Büchern von Microsoft Press:

Duncan, Ray. *Advanced MS-DOS®*. Redmond, Wash.: Microsoft Press, 1986.

Wolverton, Van. *Running MS-DOS*. 2. Ausgabe. Redmond, Wash.: Microsoft Press, 1985.

Wolverton, Van. *Supercharging MS-DOS*. Redmond, Wash.: Microsoft Press, 1986.

## Die Gliederung der Kapitel

Jedes Kapitel umfaßt ein allgemeines Programmierthema. Innerhalb eines jeden Kapitels führt der Stoff von grundlegenden zu erweiterten Konzepten. Beispielsweise beginnt Kapitel 5, "Grafiken", mit der Beschreibung von Bildschirmpunkten und Bildschirmkoordinaten, geht weiter über das Zeichnen einfacher Formen wie Geraden und Kreisen und endet schließlich mit der Beschreibung von Zeichentrickeffekten.

Wenn Sie ein erfahrener BASIC-Programmierer sind, möchten Sie vielleicht sofort in die speziellen Abschnitte innerhalb eines Kapitels springen, die Ihr besonderes Interesse haben. Wenn Sie jedoch mit einem gegebenen Thema nicht vertraut sind, werden Sie es wahrscheinlich vorteilhaft finden, jedes Kapitel von Anfang bis Ende zu lesen.

### Anwendungsbeispiele

Am Ende eines jeden Kapitels befinden sich unter der Überschrift "Anwendungsbeispiele" ein oder mehrere Listing/s vollständiger, funktionierender Programme. Jedes dieser Anwendungsbeispiele befindet sich ebenso auf Ihren QuickBASIC-Originaldisketten. Nachdem Sie die Listings im Text studiert haben, können Sie sofort sehen wie diese Programme arbeiten, ohne sie selbst eintippen zu müssen. Der Dateiname erscheint in der Überschrift für jedes Programm.

Einige der längeren Beispielprogramme, die sich über alle Kapitel erstrecken, befinden sich ebenfalls auf Diskette. Die Dateinamen dieser Programme sind immer in den einleitenden Bemerkungen aufgeführt.

### Zusammenfassende Tabellen

Jedes Kapitel endet mit einer Tabelle, die die in dem Kapitel erläuterten BASIC-Anweisungen und -Funktionen zusammenfaßt. Diese Anweisungen und Funktionen sind nach Programmieraufgaben geordnet, wie z. B. das Erhalten von Informationen über eine Datendatei oder das Benutzen von Farbe bei Grafikausgaben.

### Syntax der Anweisungen und Funktionen

Wenn die Syntax für eine bestimmte Anweisung oder Funktion eingeführt wird, ist sie oft in einem vereinfachten Format ohne jede Option oder Nuance gegeben. Zum Teil werden beschreibende Platzhalter verwendet. Eine umfassende Beschreibung der Syntax der Anweisung oder Funktion finden Sie im *BASIC-Befehlsverzeichnis*.

## Typographische Konventionen

In diesem Handbuch werden folgende typographische Konventionen verwendet:

| *Darstellung* | *Beschreibung* |
|---|---|
| `Beispiele` **`Eingabe:`** | Die in der linken Spalte gezeigte Schriftart wird zur Darstellung von Informationen benutzt, die auf Ihrem Bildschirm erscheinen. Die fettgedruckte Version dieses Schrifttyps zeigt an, daß eine Eingabe nach einem Eingabeaufforderungszeichen eingegeben wird. |
| Apostroph (') | Das Apostroph kennzeichnet einen Kommentar in Beispielprogrammen. Ein Apostroph wird durch Drücken der Taste eingegeben, die ein einfaches rechtes Anführungszeichen erzeugt. Die Taste kann als ' oder " beschriftet sein. Benutzen Sie nicht das einfache linke Anführungszeichen, welches durch die mit ` gekennzeichnete Taste erzeugt wird. |
| **SCHLÜSSELWÖRTER** | Fettgedruckte Großbuchstaben zeigen BASIC-Schlüsselwörter an. Diese Schlüsselwörter sind ein nötiger Bestandteil der Anweisungs-Syntax, es sei denn, sie sind, wie nachfolgend beschrieben, in eckige Klammern eingeschlossen. Wenn Sie Programme schreiben, müssen Sie Schlüsselwörter genau wie gezeigt eingeben. Sie können dabei Groß- und/oder Kleinbuchstaben benutzen. |
| **Befehlsnamen** | Fettgedruckte Kleinbuchstaben im Text zeigen Menü- und Befehlsnamen an. |
| *Platzhalter* | Die Begriffe in Kursivschrift sind Platzhalter für Informationen, die Sie angeben müssen, wie z. B. einen Dateinamen.<br>Darüber hinaus wird Kursivschrift gelegentlich im Text zur Hervorhebung benutzt. |

| *Darstellung* | *Beschreibung* |
|---|---|
| **[Wahlweise Begriffe]** | Zwischen eckigen Klammern stehende Begriffe können wahlweise verwendet werden. |
| {*Wahl1* \| *Wahl2*} | Geschweifte Klammern und ein vertikaler Balken zeigen an, daß Sie die Wahl zwischen zwei oder mehr Begriffen haben. Sie müssen einen der Begriffe wählen, es sei denn, alle Begriffe sind auch in eckige Klammern eingeschlossen. |
| Wiederholende Elemente... | Drei Punkte, die einem Begriff folgen, zeigen an, daß mehrere Begriffe derselben Form auftreten können. |
| `Programm`<br>.<br>.<br>.<br>`Fragment` | Eine senkrechte Punktreihe zeigt an, daß ein Teil des Programms weggelassen wurde. |
| TASTENBEZEICHNUNGEN | Kleingedruckte Großbuchstaben werden für die Bezeichnungen von Tasten und Tastenkombinationen verwendet, wie z.B. EINGABETASTE und STRG+R. Beachten Sie bitte, daß ein Plus (+) eine Kombination von Tasten anzeigt. Zum Beispiel zeigt Ihnen STRG+E, daß die STRG-TASTE gedrückt bleiben muß, während die E-Taste betätigt wird.<br>Die in diesem Handbuch verwendeten Tastenbezeichnungen entsprechen den Bezeichnungen, die auf den Tasten des IBM®-Personal Computers stehen. Falls Sie einen anderen Computer benutzen, können diese Tasten abweichende Bezeichnungen tragen.<br>Die Gruppe der Cursor-Bewegungstasten (auch "Pfeiltasten" genannt), die sich im numerischen Tastenfeld rechts neben der Haupttastatur befinden, werden RICHTUNGSTASTEN genannt. Jede einzelne RICHTUNGSTASTE wird entweder durch die Richtung des Pfeiles auf der Taste bezeichnet (NACH LINKS, NACH RECHTS, NACH OBEN, NACH UNTEN) oder durch die Bezeichnung auf der Taste (BILD ↑, BILD ↓).<br>Die Wagenrücklauftaste wird als EINGABETASTE bezeichnet. |
| "Definierter Ausdruck" | Anführungsstriche kennzeichnen entweder einen im Text definierten Ausdruck oder den Namen eines Begriffs in einem Dialogfeld. |

***Hinweis*** Die unten gezeigte Syntax (für die Anweisungen "**LOCK...UNLOCK**") verwendet viele der Darstellungskonventionen dieses Handbuches:

**LOCK** [#]*Dateinummer*[,{*Satz* | [*Start*]**TO** *Ende*}]
.
.
.
**UNLOCK** [#]*Dateinummer*[,{*Satz* | [*Start*]**TO** *Ende*}]

## Der in diesem Handbuch verwendete Programmierstil

Es folgen die Richtlinien, die beim Schreiben der Programme, die sich sowohl im Handbuch als auch auf den Originaldisketten befinden, verwendet wurden.

Obwohl diese Richtlinien für die Programmlesbarkeit empfohlen werden, müssen Sie diese beim Schreiben eigener Programme nicht befolgen.

- Schlüsselwörter und symbolische Konstanten werden in Großbuchstaben dargestellt:

```
' PRINT, DO, LOOP, UNTIL sind Schlüsselworte:
PRINT "Titelseite"
DO LOOP UNTIL Antwort$ = "N"
' FALSCH und WAHR sind symbolische Konstanten
' gleich 0 bzw. -1:
CONST FALSCH = 0, WAHR = NOT FALSCH
```

- Variablennamen werden in Kleinbuchstaben mit einem großen Anfangsbuchstaben dargestellt; Variablennamen mit mehr als einer Silbe können weitere Großbuchstaben enthalten, um einen Silbenanfang anzuzeigen:

```
AnzSaetze% = 45
GebDat$ = "04/09/58"
```

- Zeilenmarken werden anstelle von Zeilennummern verwendet. Das Benutzen von Zeilenmarken beschränkt sich auf Routinen zur Ereignisverfolgung und Fehlerbehandlung sowie auf **DATA**-Anweisungen, auf die mit **RESTORE** Bezug genommen wird:

```
' ZeitBehand und BildZweiDaten sind Zeilenmarken:
ON TIMER GOSUB ZeitBehand
RESTORE BildZweiDaten
```

- Ein einzelnes Apostroph (') leitet einen Kommentar ein:

```
' Dies ist ein Kommentar; diese beiden Zeilen werden
' vom Programm während der Ausführung ignoriert.
```

- Blöcke zur Ablaufsteuerung und Anweisungen in Prozeduren oder Unterroutinen werden wie folgt zwischen den umschließenden Code eingerückt:

```
SUB Eingabe STATIC
   FOR I% = 1 TO 10
      INPUT X
      IF X > 0 THEN
         .
         .
         .
      ELSE
         .
         .
         .
      END IF
   NEXT I%
END SUB
```

# 1 Strukturen zur Ablaufsteuerung


1.1 Wie Sie die Reihenfolge der Ausführung von Anweisungen ändern 1.2

1.2 Boolesche Ausdrücke 1.3

1.3 Entscheidungsstrukturen 1.6
- 1.3.1 Block-**IF...THEN...ELSE** 1.9
- 1.3.2 **SELECT CASE** 1.11
  - 1.3.2.1 Wie Sie die **SELECT CASE**-Anweisung verwenden 1.13
  - 1.3.2.2 **SELECT CASE** im Vergleich mit **ON GOSUB** 1.17

1.4 Schleifenstrukturen 1.19
- 1.4.1 **FOR...NEXT**-Schleifen 1.19
  - 1.4.1.1 Das Verlassen einer **FOR...NEXT**-Schleife mit **EXIT FOR** 1.24
  - 1.4.1.2 Verzögerung der Programmausführung mit einer **FOR...NEXT**-Schleife 1.25
- 1.4.2 **WHILE...WEND**-Schleifen 1.25
- 1.4.3 **DO...LOOP**-Schleifen 1.27
  - 1.4.3.1 Schleifenprüfungen: Eine Möglichkeit, **DO...LOOP** zu verlassen 1.30
  - 1.4.3.2 **EXIT DO**: Ein weiterer Weg, **DO...LOOP** zu verlassen 1.32
  - 1.4.3.3 Verzögerung der Programmausführung mit **DO...LOOP** 1.32

1.5 Anwendungsbeispiele 1.33
- 1.5.1 Programm zur Scheckheftbilanz (SCHECK.BAS) 1.34
- 1.5.2 Filter für Wagenrücklauf-Zeilenvorschub (WRZV.BAS) 1.35

1.6 Zusammenfassung der Anweisungen zur Ablaufsteuerung 1.40

Dieses Kapitel zeigt Ihnen, wie Sie Strukturen zur Ablaufsteuerung verwenden, besonders Schleifen und Entscheidungsanweisungen, um den Ablauf der Ausführung Ihres Programmes zu steuern. Schleifen veranlassen ein Programm, eine Reihe von Anweisungen sooft zu wiederholen, wie Sie wollen. Entscheidungsanweisungen lassen das Programm entscheiden, welcher Pfad aus verschiedenen Pfaden ausgewählt werden soll.

Am Ende dieses Kapitels werden Sie wissen, wie Sie die folgenden Aufgaben im Zusammenhang mit dem Einsatz von Schleifen und Entscheidungsanweisungen in Ihren BASIC-Programmen bearbeiten können:

- Vergleichen von Ausdrücken mit Vergleichsoperatoren.
- Kombinieren von Zeichenketten- oder numerischen Ausdrücken mit logischen Operatoren und Feststellen, ob der resultierende Ausdruck wahr oder falsch ist.
- Erstellen von Verzweigungen in dem Ablauf des Programmes mit den Anweisungen **IF...THEN...ELSE** und **SELECT CASE.**
- Schreiben von Schleifen, die Anweisungen sooft wie angegeben wiederholen.
- Schreiben von Schleifen, die Anweisungen wiederholen, solange oder bis eine bestimmte Bedingung wahr ist.

## 1.1 Wie Sie die Reihenfolge der Ausführung von Anweisungen ändern

Ein Anweisungsblock besteht aus einer oder mehreren zusammenstehenden Anweisungen auf einer oder mehreren Zeile/n. Wenn die Logik eines Programmes nicht von Anweisungen zur Ablaufsteuerung kontrolliert wird, durchläuft die Programmlogik Anweisungsblöcke von links nach rechts und von oben nach unten. Während einige sehr einfache Programme mit diesem geradlinigen Ablauf geschrieben werden können, ergibt sich die eigentliche Stärke und Verwendbarkeit einer Programmiersprache aus ihrer Fähigkeit, die Reihenfolge der Ausführung von Anweisungen durch Entscheidungsstrukturen und Schleifen zu ändern.

Mit einer Entscheidungsstruktur kann ein Programm einen Ausdruck auswerten und anschließend abhängig vom Ergebnis dieser Auswertung in verschiedene Anweisungsblöcke verzweigen. Mit einer Schleife kann ein Programm Anweisungsblöcke wiederholt ausführen.

Falls Sie schon vorher in BASICA programmiert haben, werden Sie die zusätzliche Flexibilität der folgenden Strukturen zur Ablaufsteuerung in dieser BASIC-Version zu schätzen wissen:

- Die Block-**IF...THEN...ELSE**-Anweisung
- Die **SELECT CASE**-Anweisung
- Die **DO LOOP**- und **EXIT DO**-Anweisungen
- Die **EXIT FOR**-Anweisung, die eine Alternative zum Verlassen von **FOR...NEXT**-Schleifen bietet

## 1.2 Boolesche Ausdrücke

Boolesche Ausdrücke erscheinen sowohl in Entscheidungsstrukturen als auch in Schleifen. Einfach gesagt ist ein Boolescher Ausdruck jeder Ausdruck, der den Wert "wahr" oder "falsch" angibt. Wie in Abschnitt 1.3 und 1.4 beschrieben, verwendet BASIC Boolesche Ausdrücke in bestimmten Arten von Entscheidungstrukturen und Schleifen. Die folgende **IF...THEN...ELSE**-Anweisung enthält einen Booleschen Ausdruck, `X < Y`:

```
IF X < Y THEN CALL Prozedur1 ELSE CALL Prozedur2
```

Wenn in diesem Beipiel der Boolesche Ausdruck wahr ist (dies bedeutet, daß der Wert der Variablen `X` tatsächlich kleiner als der Wert der Variablen `Y` ist), wird `Prozedur1` ausgeführt; anderenfalls (`X` ist größer oder gleich `Y`) wird `Prozedur2` ausgeführt.

Das vorhergehende Beispiel zeigt eine übliche Verwendung Boolescher Ausdrücke: zwei verschiedene Ausdrücke werden verglichen (in diesem Fall `X` und `Y`), um deren Beziehung zueinander zu bestimmen. Diese Vergleiche werden mit den in der folgenden Tabelle gezeigten Vergleichsoperatoren durchgeführt.

| *Operator* | *Bedeutung* |
|---|---|
| = | gleich |
| <> | ungleich |
| < | kleiner als |
| <= | kleiner gleich |
| > | größer als |
| >= | größer gleich |

Sie können diese Vergleichsoperatoren zum Vergleichen von Zeichenkettenausdrücken verwenden. In diesem Fall beziehen sich "größer", "kleiner" etc. auf die alphabetische Reihenfolge. Zum Beispiel ist der folgende Ausdruck wahr, da das Wort "andere" in der alphabetischen Reihenfolge vor dem Wort "anders" liegt:

```
"andere" < "anders"
```

Boolesche Ausdrücke verwenden auch häufig die "logischen Operatoren" **AND**, **OR**, **NOT**, **XOR**, **IMP** und **EQV**. Diese Operatoren erlauben es Ihnen, zusammengesetzte Prüfungen mit einem oder mehreren Booleschen Ausdrücken aufzubauen. Zum Beispiel ist

*Ausdruck1* **AND** *Ausdruck2*

nur dann wahr, wenn *Ausdruck1* und *Ausdruck2* beide wahr sind. Daraus folgt, daß die Meldung `Alles sortiert` im folgenden Beispiel nur dann ausgegeben wird, wenn beide der Booleschen Ausdrücke `X <= Y` und `Y <= Z` wahr sind:

```
IF (X <= Y) AND (Y <= Z) THEN PRINT "Alles sortiert"
```

Im letzten Beispiel sind die Klammern um den Booleschen Ausdruck nicht unbedingt notwendig, da Vergleichsoperatoren wie <= vor logischen Operatoren wie **AND** ausgewertet werden. Klammern können jedoch hilfreich sein, einen komplexen Booleschen Ausdruck lesbarer zu gestalten und sicherzustellen, daß seine Komponenten in der von Ihnen gewünschten Reihenfolge ausgewertet werden.

BASIC verwendet die numerischen Werte -1 und 0, um wahr und falsch darzustellen. Sie können dies sehen, wenn Sie BASIC, wie im nächsten Beispiel gezeigt, veranlassen, einen wahren Ausdruck und einen falschen Ausdruck auszugeben:

```
X = 5
Y = 10
PRINT X < Y      'Auswertung und Ausgabe eines wahren
                 'Booleschen Ausdrucks
PRINT X > Y      'Auswertung und Ausgabe eines falschen
                  Booleschen Ausdrucks
```

### Ausgabe

```
-1
 0
```

Der Wert -1 für wahr wird einleuchtender, wenn Sie die Arbeitsweise des **NOT**-Operators von BASIC betrachten: **NOT** invertiert jedes Bit der binären Darstellung seines Operanden, indem 1-Bits in 0-Bits geändert werden und 0-Bits zu 1-Bits. Da der ganzzahlige Wert 0 (falsch) intern als eine Folge von sechzehn 0-Bits gespeichert wird, wird **NOT** 0 (wahr) intern als sechzehn 1-Bits gespeichert, wie unten gezeigt:

```
FALSCH              = 0000000000000000
WAHR = NOT FALSCH   = 1111111111111111
```

In der Zweierkomplementmethode, die BASIC zum Speichern von Ganzzahlen verwendet, stellen sechzehn 1-Bits den Wert -1 dar.

***Wichtig*** Beachten Sie, daß BASIC selbst -1 ausgibt, wenn es einen Booleschen Ausdruck als wahr auswertet. BASIC betrachtet jedoch *jeden* Wert ungleich null als wahr, wie in der Ausgabe des folgenden Beispiels gezeigt:

```
INPUT "Geben Sie einen Wert ein: ", X
IF X THEN PRINT X "ist wahr"
```

***Ausgabe***

```
Geben Sie einen Wert ein: 2
 2 ist wahr
```

In BASIC ist der **NOT**-Operator als "bitweiser" Operator bekannt. Einige Programmiersprachen, wie C oder Pascal, haben sowohl einen bitweisen **NOT**-Operator als auch einen "logischen" **NOT**-Operator. Der Unterschied ist wie folgt:

- Ein bitweises **NOT** gibt falsch (0) nur für den Wert -1 aus.
- Ein logisches **NOT** gibt falsch (0) für jeden wahren (ungleich null) Wert aus.

Wie in der folgenden Liste aufgeführt, gibt in BASIC **NOT** *Ausdruck* für jeden wahren *Ausdruck* ungleich -1 einen anderen wahren Wert an:

| ***Wert des Ausdrucks*** | ***Wert des NOT-Ausdrucks*** |
|---|---|
| 1 | -2 |
| 2 | -3 |
| -2 | 1 |
| -1 | 0 |

Beachten Sie also: **NOT** *Ausdruck* ist nur falsch, wenn *Ausdruck* einen Wert von -1 ergibt. Wenn Sie Boolesche Konstanten oder Variablen für die Verwendung in Ihrem Programm definieren, sollten Sie -1 für wahr einsetzen.

Sie können die Werte 0 und -1 verwenden, um hilfreiche mnemonische Boolesche Konstanten zur Verwendung in Schleifen oder Entscheidungen zu definieren. Diese Technik wird in vielen der Beispiele dieses Handbuches verwendet, wie in dem folgenden Programmausschnitt gezeigt ist, der die Elemente eines Datenfeldes `Betrag` in aufsteigender Reihenfolge sortiert:

```
' Definiere symbolische Konstanten zur Verwendung im
' Programm:
CONST FALSCH = 0, WAHR = NOT FALSCH
.
.
.
DO
   Tausch = FALSCH
   FOR I = 1 TO TransakAnz-1
      IF Betrag(I) < Betrag(I+1) THEN
         SWAP Betrag(I), Betrag(I+1)
         Tausch = WAHR
      END IF
   NEXT I
LOOP WHILE Tausch       ' Durchlaufe die Schleife
                        ' solange Tausch WAHR ist
.
.
.
```

## 1.3 Entscheidungsstrukturen

Ausgehend von dem Wert eines Ausdrucks veranlassen Entscheidungsstrukturen ein Programm dazu, eine der folgenden Aktionen durchzuführen:

1. Ausführen einer von verschiedenen alternativen Anweisungen innerhalb der Entscheidungsstruktur selbst.
2. Verzweigen in einen anderen Teil des Programmes außerhalb der Entscheidungsstruktur.

In BASICA werden Entscheidungen nur mit der einzeiligen **IF...THEN[...ELSE]**-Anweisung bearbeitet. In ihrer einfachsten Form (**IF...THEN**) wird der Ausdruck ausgewertet, der dem Schlüsselwort **IF** folgt. Ist dieser Ausdruck wahr (ungleich 0), führt das Programm die Anweisungen aus, die dem Schlüsselwort **THEN** folgen; wenn der Ausdruck falsch (0) ist, fährt das Programm mit der nächsten Zeile nach der **IF...THEN**-Anweisung fort. Die Zeilen 50 und 70 des folgenden BASICA-Programmausschnittes zeigen Beispiele für **IF...THEN**:

```
20  INPUT A
30  ' Wenn A größer als 100 ist, gib eine Meldung aus
35  ' und gehe zurück zu Zeile 20; andernfalls gehe
40  ' weiter zu Zeile 60:
50  IF A > 100 THEN PRINT "Zu groß": GOTO 20
60  ' Wenn A gleich 100 ist, springe in Zeile 300,
65  ' anderenfalls gehe zu Zeile 80:
70  IF A = 100 THEN GOTO 300
80  PRINT A/100 : GOTO 20
.
.
.
```

Durch Hinzufügen der **ELSE**-Klausel zu einer **IF...THEN**-Anweisung können Sie Ihr Programm veranlassen, eine Reihe von Funktionen (diejenigen, die dem Schlüsselwort **THEN** folgen) durchzuführen, wenn ein Ausdruck wahr ist, und eine andere Reihe von Funktionen (diejenigen, die dem Schlüsselwort **ELSE** folgen), wenn er falsch ist. Der nächste Programmausschnitt zeigt, wie **ELSE** in einer **IF...THEN...ELSE**-Anweisung funktioniert.

```
10 INPUT "Geben Sie Ihre Kennung ein"; Kenn$
15 ' Wenn der Benutzer "Britta" eingibt, gehe zu
20 ' Zeile 50, andernfalls gib eine Meldung aus und
25 ' gehe zurück zu Zeile 10.
30 IF Kenn$ = "Britta" THEN
31    50
32 ELSE
33     PRINT "Versuchen Sie noch einmal" : GOTO 10
34 END IF
.
.
.
```

Während das einzeilige **IF...THEN...ELSE** des BASICA für einfache Entscheidungen angemessen ist, kann es zu wirklich unlesbarem Code führen, wenn Sie komplizierte Entscheidungen formulieren wollen. Dies ist besonders dann der Fall, wenn Sie Ihre Programme so gestalten, daß alle alternativen Funktionen innerhalb der **IF...THEN...ELSE**-Anweisung selbst ausgeführt werden, oder wenn Sie **IF...THEN...ELSE**-Anweisungen verschachteln (das heißt, wenn Sie eine **IF...THEN...ELSE** innerhalb einer anderen plazieren, eine durchaus legale Konstruktion). Betrachten Sie den nächsten Ausschnitt eines BASICA-Programmes als ein Beispiel, wie schwierig es ist, selbst einer einfachen Prüfung zu folgen:

```
10 ' Die folgenden verschachtelten IF...THEN...ELSE-
15 ' Anweisungen schreiben verschiedene Ausgaben für
20 ' jeden der vier folgenden Fälle:
25 '   1) A <= 50, B <= 50     3)  A > 50, B <= 50
30 '   2) A <= 50, B > 50      4)  A > 50, B > 50
35 '
40 INPUT A, B
45 '
50 ' Beachte: Obwohl Zeile 80 über mehrere
55 ' physikalische Zeilen des Bildschirms geht, ist
60 ' sie nur eine "logische Zeile" (alles, was vor
65 ' Betätigung der Eingabetaste eingetippt wurde).
70 ' BASICA bricht lange Zeilen auf dem Bildschirm um.
75 '
80  IF A <= 50 THEN IF B <= 50 THEN PRINT "A <= 50, B<= 50"
ELSE PRINT "A <= 50, B > 50" ELSE IF B <= 50    THEN PRINT
"A > 50, B <= 50" ELSE PRINT "A > 50, B >    50"
```

Um diese oben gezeigte Art komplizierter Anweisungen zu vermeiden, enthält BASIC nun die Blockform der **IF...THEN...ELSE**-Anweisung, so daß eine Entscheidung nicht länger auf eine logische Zeile begrenzt ist. Das folgende Beispiel zeigt das vorhergehende BASICA-Programm neu geschrieben unter Verwendung des Block-**IF...THEN...ELSE**:

```
INPUT A, B
IF A <= 50 THEN
   IF B <= 50 THEN
      PRINT "A <= 50, B <= 50"
   ELSE
      PRINT "A <= 50, B > 50"
   END IF
```

```
ELSE
   IF B <= 50 THEN
      PRINT "A > 50, B <= 50"
   ELSE
      PRINT "A > 50, B > 50"
   END IF
END IF
```

Weitere Informationen zu Block-**IF...THEN...ELSE** finden Sie in Abschnitt 1.3.1.

Für strukturierte Entscheidungen kennt QuickBASIC ebenso die Anweisung **SELECT CASE...END SELECT** (im folgenden als **SELECT CASE** bezeichnet).

Sowohl die Block-**IF...THEN...ELSE**-Anweisung als auch die **SELECT CASE**-Anweisung erlauben es Ihnen, Ihren Programmcode nach logischen Gesichtspunkten zu gestalten und nicht nach der Anzahl der Anweisungen, die auf einer Zeile untergebracht werden können. Dieses bietet Ihnen nicht nur den Vorteil erhöhter Flexibilität während der Programmierung, sondern ebenso den Vorteil verbesserter Programmlesbarkeit und leichterer Wartung.

## 1.3.1 Block-IF...THEN...ELSE

Die folgende Tabelle zeigt die Syntax der Block-**IF...THEN...ELSE**-Anweisung sowie ein Beispiel für ihre Verwendung.

| ***Syntax*** | ***Beispiel*** |
|---|---|
| **IF** *Bedingung1* **THEN**<br>[*Anweisungsblock-1*]<br>[**ELSEIF** *Bedingung2* **THEN**<br>[*Anweisungsblock-2*]]<br>.<br>.<br>.<br>[**ELSE**<br>[*Anweisungsblock-n*]]<br>**END IF** | `IF X > 0 THEN`<br>`  PRINT "X ist positiv"`<br>`  PosZahl = PosZahl + 1`<br>`ELSEIF X < 0 THEN`<br>`  PRINT "X ist negativ"`<br>`  NegZahl = NegZahl + 1`<br>`ELSE`<br>`  PRINT "X ist Null"`<br>`END IF` |

Die Argumente *Bedingung1*, *Bedingung2* usw. sind Ausdrücke. Sie können jeder numerische Ausdruck-in diesem Fall ist wahr jeder Wert ungleich null und falsch ist null- oder sie können Boolesche Ausdrücke sein, in diesem Fall ist wahr -1 und falsch ist null. Wie in Abschnitt 1.2 erläutert, vergleichen Boolesche Ausdrücke typischerweise zwei numerische oder Zeichenkettenausdrücke, indem sie einen der Vergleichsoperatoren wie <, > oder = verwenden.

Jeder **IF-**, **ELSEIF-** und **ELSE**-Klausel folgt ein Anweisungsblock. Keine der Anweisungen in dem Block kann sich auf der gleichen Zeile wie die **IF-**, **ELSEIF-** oder **ELSE**-Klausel befinden. Anderenfalls faßt BASIC sie als einzeilige **IF THEN**-Anweisung auf.

BASIC wertet jeden der Ausdrücke in den **IF-** und **ELSEIF**-Klauseln von oben nach unten aus und übergeht Anweisungsblöcke, bis es den ersten wahren Ausdruck findet. Wenn es einen wahren Ausdruck findet, führt es die zu dem Ausdruck gehörenden Anweisungen aus und verzweigt dann aus dem Block heraus zu der Anweisung, die der **END IF**-Klausel folgt.

Wenn keiner der Ausdrücke in der **IF-** oder **ELSEIF**-Klausel wahr ist, springt BASIC zu der **ELSE**-Klausel, wenn eine solche existiert, und führt deren Anweisungen aus. Wenn jedoch keine **ELSE**-Klausel vorhanden ist, fährt das Programm mit der nächsten Anweisung nach der **END IF**-Klausel fort.

Die Klauseln **ELSE** und **ELSEIF** sind beide optional, wie im folgenden Beispiel gezeigt:

```
' Wenn der Wert von X kleiner als 100 ist, führe die
' beiden Anweisungen vor END IF aus; anderenfalls gehe
' zu der INPUT-Anweisung, die nach END IF steht:
IF X < 100 THEN
   PRINT X
   Nummer = Nummer + 1
END IF
INPUT "Neuer Wert"; Antwort$
.
.
.
```

Ein einzelnes Block-**IF...THEN...ELSE** kann mehrere **ELSEIF**-Anweisungen enthalten, wie unten gezeigt:

```
IF C$ >= "A" AND C$ <= "Z" THEN
   PRINT "Großbuchstaben."
ELSEIF C$ >= "a" AND C$ <= "z" THEN
   PRINT "Kleinbuchstaben."
ELSEIF C$ >= "0" AND C$ <= "9" THEN
   PRINT "Zahl."
ELSE
   PRINT "Nicht alphanumerisch."
END IF
```

Es wird höchstens ein Block von Anweisungen ausgeführt, selbst wenn mehr als eine Bedingung wahr ist. Wenn Sie beispielsweise das Wort `als` in die Eingabe des nächsten Beispiels schreiben, erscheint die Meldung `Eingabe zu kurz`, es erscheint jedoch nicht die Meldung `Kann nicht mit a beginnen:`

```
INPUT Check$
IF LEN(Check$) > 6 THEN
   PRINT "Eingabe zu lang"
ELSEIF LEN(Check$) < 6 THEN
   PRINT "Eingabe zu kurz"
ELSEIF LEFT$(Check$, 1) = "a" THEN
   PRINT "Kann nicht mit a beginnen"
END IF
```

**IF...THEN...ELSE**-Anweisungen können verschachtelt werden; mit anderen Worten, eine **IF...THEN...ELSE**-Anweisung kann weitere **IF...THEN...ELSE**-Anweisungen enthalten, wie hier gezeigt:

```
IF X > 0 THEN
   IF Y > 0 THEN
      IF Z > 0 THEN
         PRINT "Alle größer als Null."
      ELSE
         PRINT "Nur X und Y sind größer als Null."
      END IF
   END IF
ELSEIF X = 0 THEN
   IF Y = 0 THEN
      IF Z = 0 THEN
         PRINT "Alle gleich Null."
      ELSE
         PRINT "Nur X und Y sind gleich Null."
      END IF
   END IF
ELSE
   PRINT "X ist kleiner als Null."
END IF
```

## 1.3.2 SELECT CASE

Die **SELECT CASE**-Anweisung ist eine Entscheidungsstruktur mit mehreren Möglichkeiten, die der Block-**IF...THEN...ELSE**-Anweisung ähnlich ist. Block-**IF..THEN...ELSE** kann überall benutzt werden, wo auch **SELECT CASE** verwendet wird.

Der entscheidende Unterschied zwischen den beiden besteht darin, daß **SELECT CASE** einen *einzigen* Ausdruck auswertet, dann abhängig vom Ergebnis verschiedene Anweisungen ausführt oder zu verschiedenen Teilen des Programmes verzweigt. Ein Block-**IF...THEN...ELSE** dagegen kann völlig verschiedene Ausdrücke auswerten.

### Beispiele

Die folgenden Beispiele zeigen die Ähnlichkeiten und die Unterschiede zwischen **SELECT CASE**- und **IF...THEN...ELSE**-Anweisungen.

Das nächste Beispiel zeigt eine Entscheidung mit mehreren Möglichkeiten, die Block-**IF...THEN...ELSE** verwendet:

```
INPUT X
IF X = 1 THEN
   PRINT "eins"
ELSEIF X = 2 THEN
   PRINT "zwei"
ELSEIF X = 3 THEN
   PRINT "drei"
ELSE
   PRINT "muß ganzzahlig zwischen 1-3 sein"
END IF
```

Die obige Entscheidung ist im nächsten Beispiel neu geschrieben mit **SELECT CASE**:

```
INPUT X
SELECT CASE x
   CASE 1
      PRINT "eins"
   CASE 2
      PRINT "zwei"
   CASE 3
      PRINT "drei"
   CASE ELSE
      PRINT "muß ganzzahlig zwischen 1-3 sein"
END SELECT
```

Die folgende Entscheidung kann mit der Block-**IF...THEN...ELSE**-Anweisung rationeller bearbeitet werden, da in der **IF**- und **ELSEIF**-Klausel verschiedene Ausdrücke ausgewertet werden:

```
INPUT X, Y
IF X = 0 AND Y = 0 THEN
   PRINT "beide Null."
ELSEIF X = 0 THEN
   PRINT "nur X ist Null."
ELSEIF Y = 0 THEN
   PRINT "nur Y ist Null."
ELSE
   PRINT "keine von beiden ist Null."
END IF
```

### 1.3.2.1 Wie Sie die SELECT CASE-Anweisung verwenden

Die folgende Tabelle zeigt die Syntax der **SELECT CASE**-Anweisung und ein Beispiel ihrer Verwendung.

| *Syntax* | *Beispiel* |
|---|---|
| **SELECT CASE** *Ausdruck*<br>**CASE** *Ausdrucksliste1*<br>[*Anweisungsblock-1*]<br>[**CASE** *Ausdrucksliste2*<br>[*Anweisungsblock-2*]]<br>.<br>.<br>.<br>[**CASE ELSE**<br>[*Anweisungsblock-n*]]<br>**END SELECT** | `INPUT TestWert`<br>`SELECT CASE TestWert`<br>`  CASE 1,3,5,7,9`<br>`  PRINT "ungerade"`<br>`  CASE 2,4,6,8`<br>`  PRINT "gerade"`<br>`  CASE IS < 1`<br>`  PRINT "zu klein"`<br>`  CASE IS > 9`<br>`  PRINT "zu groß"`<br>`  CASE ELSE`<br>`  PRINT "keine Ganzzahl"`<br>`END SELECT` |

Die Argumente der *Ausdrucksliste*, die der **CASE**-Klausel folgen, können eine oder mehrere der folgenden, durch Kommata getrennten Argumente sein:

- Ein numerischer Ausdruck oder ein Bereich von numerischen Ausdrücken
- Ein Zeichenkettenausdruck oder ein Bereich von Zeichenkettenausdrücken

Um einen Bereich von Ausdrücken festzulegen, benutzen Sie die folgende Syntax für die **CASE**-Anweisung:

**CASE** *Ausdruck* **TO** *Ausdruck*
**CASE IS** *Vergleichsoperator Ausdruck*

wobei *Vergleichsoperator* ein beliebiger Operator aus der Tabelle in Abschnitt 1.2 sein kann.

Wenn Sie beispielsweise `CASE 1 TO 4` benutzen, werden die mit diesem Fall verknüpften Anweisungen ausgeführt, wenn der *Ausdruck* in der **SELECT CASE**-Anweisung größer gleich eins und kleiner gleich 4 ist. Wenn Sie `CASE IS < 5` benutzen, werden die entsprechenden Anweisungen nur ausgeführt, wenn *Ausdruck* kleiner fünf ist.

Wenn Sie einen Bereich mit dem Schlüsselwort **TO** ausdrücken, müssen Sie den kleineren Wert zuerst schreiben. Wenn Sie beispielsweise überprüfen wollen, ob ein Wert im Bereich von `-5` bis `-1` liegt, sollten Sie die `CASE`-Anweisung wie folgt schreiben:

```
CASE -5 TO -1
```

Die folgende Anweisung ist *keine* gültige Möglichkeit, den Bereich von `-5` bis `-1` festzulegen, so daß die mit diesem Fall verknüpften Anweisungen nie ausgeführt werden:

```
CASE -1 TO -5
```

Ähnlich sollte ein Bereich von Zeichenkettenkonstanten zuerst die Zeichenkette aufführen, die alphabetisch zuerst kommt:

```
CASE "aber" TO "bunt"
```

Mehrere Ausdrücke oder Bereiche von Ausdrücken können für jede **CASE**-Klausel aufgeführt werden, wie in den folgenden Zeilen:

```
CASE 1  TO 4, 7 TO 9, JokerZeich1%, JokerZeich2%
CASE IS = Test$, IS = "Ende der Daten"
CASE IS < UnterGr, 5, 6, 12 , IS > OberGr
CASE IS < "HAN", "MAO" TO "TAO"
```

Wenn der Wert des **SELECT CASE**-Ausdruckes in der Liste erscheint, die dieser **CASE**-Klausel folgt, werden nur die einer **CASE**-Klausel zugeordneten Anweisungen ausgeführt. Die Kontrolle springt dann zu der ersten ausführbaren Anweisung, die **END SELECT** folgt, *nicht* zu dem nächsten Block von Anweisungen innerhalb der **SELECT CASE**-Struktur, wie durch die Ausgabe des nächsten Beispiels gezeigt (wenn der Benutzer die Zahl 1 eingibt):

```
INPUT x
SELECT CASE x
   CASE 1
      PRINT "eins, ";
   CASE 2
      PRINT "zwei, ";
   CASE 3
      PRINT "drei, ";
END SELECT
PRINT "das war's"
```

### *Ausgabe*

```
? 1
eins, das war's
```

Wenn derselbe Wert oder derselbe Bereich von Werten in mehr als einer **CASE**-Klausel erscheint, werden nur die Anweisungen, die mit dem ersten Vorkommen verbunden sind, ausgeführt, wie in der nächsten Ausgabe gezeigt (der Benutzer hat `ATLAS` eingegeben):

```
INPUT Test$
SELECT CASE Test$
   CASE "A" TO "AZZZZZZZZZZZZZZZZZ"
      PRINT "Ein Wort in Großbuchstaben, beginnend mit A"
   CASE IS < "A"
      PRINT "Keine alphabetischen Zeichen"
   CASE "ATLAS"
      ' Dieser Fall wird niemals ausgeführt, da das
      ' Wort ATLAS im Bereich der ersten CASE-Klausel
      ' liegt:
      PRINT "Ein besonderer Fall"
END SELECT
```

### *Ausgabe*

```
? ATLAS
Ein Wort in Großbuchstaben, beginnend mit A
```

Wenn Sie eine **CASE ELSE**-Klausel benutzen, muß diese als die letzte **CASE**-Klausel innerhalb der **SELECT CASE**-Anweisung aufgeführt sein. Die Anweisungen zwischen einer **CASE ELSE**-Klausel und einer **END SELECT**-Klausel werden nur ausgeführt, wenn der *Ausdruck* keiner anderen **CASE**-Auswahl in der **SELECT CASE**-Anweisung entspricht. Eigentlich ist es ratsam, eine **CASE ELSE**-Anweisung in den **SELECT CASE**-Block zu schreiben, um auf unvorhergesehene Werte für *Ausdruck* reagieren zu können. Wenn keine Entsprechung für *Ausdruck* in einer der **CASE**-Anweisungen gefunden wird, führt dies zu der Fehlermeldung `CASE ELSE erwartet.` Beispielsweise würde der Benutzer diese Fehlermeldung erhalten, wenn er bei dem vorhergehenden Programmausschnitt das Wort `STRAUSS` eingegeben hätte.

Ein **SELECT CASE**-Block *muß* mit einer **END SELECT**-Anweisung abgeschlossen werden.

### Beispiel

Der folgende Programmausschnitt zeigt eine übliche Verwendung für die **SELECT CASE**-Anweisung: Er schreibt ein Menü auf den Bildschirm und verzweigt dann abhängig von der vom Benutzer eingetippten Zahl in verschiedene Unterprogramme.

```
DO              ' Starte Menüschleife.
   CLS          ' Bildschirm löschen.
   ' Schreibe fünf Menüpunkte auf den Bildschirm:
   PRINT "HAUPTMENÜ" : PRINT
   PRINT "1)  Hinzufügen neuer Namen"
   PRINT "2)  Löschen von Namen"
   PRINT "3)  Informationen verändern"
   PRINT "4)  Auflisten der Namen"
   PRINT
   PRINT "5)  ENDE"
   ' Gib Eingabeaufforderung aus:
   PRINT : PRINT "Geben Sie Ihre Wahl ein: (1 bis 5):"
   ' Warte auf die Eingabe des Benutzers. INPUT$(1)
   ' Liest ein eingegebenes Zeichen von der Tastatur:
   Bu$ = INPUT$(1)
   ' Benutze SELECT CASE, um Antwort zu verarbeiten:
   SELECT CASE Bu$
      CASE "1"
         CALL HinzDaten
      CASE "2"
         CALL LoeschDat
```

```
      CASE "3"
         CALL VerDaten
      CASE "4"
         CALL ListDaten
      CASE "5"
         EXIT DO      ' Die einzige Möglichkeit, die
                      ' Menüschleife zu verlassen.
      CASE ELSE
         BEEP
   END SELECT
LOOP                  ' Ende der Menüschleife.
END
' Unterprogramme HinzDaten, LoeschDat, VerDaten,
' ListDaten:
   .
   .
   .
```

### 1.3.2.2 SELECT CASE im Vergleich mit ON...GOSUB

Sie können die vielseitigere **SELECT CASE**-Anweisung als Ersatz für die alte **ON** *Ausdruck* {**GOSUB** | **GOTO**}-Anweisung verwenden. Die **SELECT CASE**-Anweisung hat folgende Vorteile gegenüber der **ON** *Ausdruck* {**GOSUB** | **GOTO**}-Anweisung:

- Der *Ausdruck* in **SELECT CASE** *Ausdruck* kann sowohl einen Zeichenkettenwert als auch einen numerischen Wert haben. Der *Ausdruck* in der **ON** *Ausdruck* {**GOSUB** | **GOTO**}-Anweisung kann nur einen numerischen Wert im Bereich von 0 bis 255 haben.
- Die **SELECT CASE**-Anweisung verzweigt zu einem Anweisungsblock, der direkt der passenden **CASE**-Klausel folgt. Im Unterschied dazu verzweigt **ON** *Ausdruck* **GOSUB** zu einer Unterroutine in einem anderen Programmteil.
- **CASE**-Klauseln können dazu verwendet werden, einen *Ausdruck* daraufhin zu überprüfen, ob er in einem Bereich von Werten liegt. Besonders wenn der Bereich sehr groß ist, kann dies mit **ON** *Ausdruck* {**GOSUB** | **GOTO**} nur schwer erreicht werden, wie im folgenden Beispiel gezeigt.

Im folgenden Ausschnitt verzweigt die **ON...GOSUB**-Anweisung abhängig vom Wert der Benutzereingabe in eine der Unterroutinen 50, 100 oder 150.

```
X% = -1
WHILE X%
   INPUT "Wählen Sie (0 für Ende): ", X%
   IF X% = 0 THEN END
   WHILE X% < 1 OR X% > 12
      PRINT "Es muß ein Wert von 1 bis 12 sein"
      INPUT "Wählen Sie (0 für Ende): ", X%
   WEND
   ON x% GOSUB 50,50,50,50,50,50,50,50,100,100,100,150
WEND
.
.
.
```

Vergleichen Sie das obige Beispiel mit dem nächsten, das eine **SELECT CASE**-Anweisung mit Wertebereichen in jeder **CASE**-Klausel verwendet:

```
DO
   INPUT "Wählen Sie (0 für Ende): ", X%
   SELECT CASE X%
      CASE 0
         END
      CASE 1 TO 8       ' Ersetzt die Anweisungen der
                        ' "Unterroutine 50" im
                        ' vorhergehenden Beispiel
      CASE 9 TO 11      ' Ersetzt die Anweisungen der
                        ' "Unterroutine 100" im
                        ' vorhergehenden Beispiel
      CASE 12           ' Ersetzt die Anweisungen der
                        ' "Unterroutine 150" im
                        ' vorhergehenden Beispiel
      CASE ELSE         ' Eingabe war außerhalb des Bereiches.
      PRINT "Es muß ein Wert von 1 bis 12 sein"
   END SELECT
LOOP
```

# 1.4 Schleifenstrukturen

Schleifenstrukturen wiederholen einen Block von Anweisungen (die Schleife) entweder sooft wie angegeben oder bis ein bestimmter Ausdruck (die Schleifenbedingung) wahr oder falsch ist.

Benutzer von BASICA sind mit den zwei Schleifenstrukturen, **FOR...NEXT** und **WHILE...WEND**, vertraut, die in den Abschnitten 1.4.1 und 1.4.2 erläutert werden. QuickBASIC hat diese dem Programmierer zur Verfügung stehenden Schleifenstrukturen durch die im Abschnitt 1.4.3 erläuterte **DO...LOOP**-Anweisung erweitert.

## 1.4.1 FOR...NEXT-Schleifen

Eine **FOR...NEXT**-Schleife wiederholt die in der Schleife eingeschlossenen Anweisungen sooft wie angegeben. Sie zählt von einem Startwert bis zu einem Endwert, indem ein Schleifenzähler entweder erhöht oder vermindert wird. Solange der Schleifenzähler den Endwert nicht erreicht hat, fährt die Schleife mit der Ausführung fort.

Die folgende Tabelle zeigt die Syntax der **FOR...NEXT**-Anweisung und gibt ein Beispiel für ihre Verwendung:

| *Syntax* | *Beispiel* |
|---|---|
| **FOR** *Zähler* = *Start* **TO** *Ende* [**STEP** *Schrittgröße*]<br>[*Anweisungsblock*]<br>[**EXIT FOR**]<br>**NEXT** [*Zähler* [, *Zähler*...]] | `FOR I% = 1 TO 10`<br>`  DatFeld%(I%) = I%`<br>`NEXT` |

In einer **FOR...NEXT**-Schleife hat die *Zähler*-Variable zu Beginn den Wert des Ausdrucks *Start*. Nach jeder Wiederholung der Schleife wird der Wert von *Zähler* angepaßt. Wenn Sie das wahlfreie Schlüsselwort **STEP** auslassen, ist der Standardwert für die Anpassung eins; das heißt, dem *Zähler* wird bei jeder Ausführung der Schleife der Wert Eins hinzugezählt. Wenn Sie **STEP** verwenden, wird der *Zähler* durch den Betrag von *Schrittgröße* angepaßt. Das Argument *Schrittgröße* kann jeder numerische Wert sein; wenn es negativ ist, zählt die Schleife abwärts vom *Start* bis zum *Ende*. Nachdem die *Zähler*-Variable erhöht oder vermindert ist, wird ihr Wert mit *Ende* verglichen. Die Schleife ist an diesem Punkt beendet, wenn einer der folgenden Punkte wahr ist:

- Die Schleife wird hochgezählt (*Schrittgröße* ist positiv) und *Zähler* ist größer als *Ende*.
- Die Schleife wird abwärts gezählt (*Schrittgröße* ist negativ) und *Zähler* ist kleiner als *Ende*.

Abbildung 1.1 zeigt die Logik einer **FOR...NEXT**-Schleife, wenn der Wert von *Schrittgröße* positiv ist.

*Abbildung 1.1 Logik einer **FOR...NEXT**-Schleife mit positiver **STEP**-Größe*

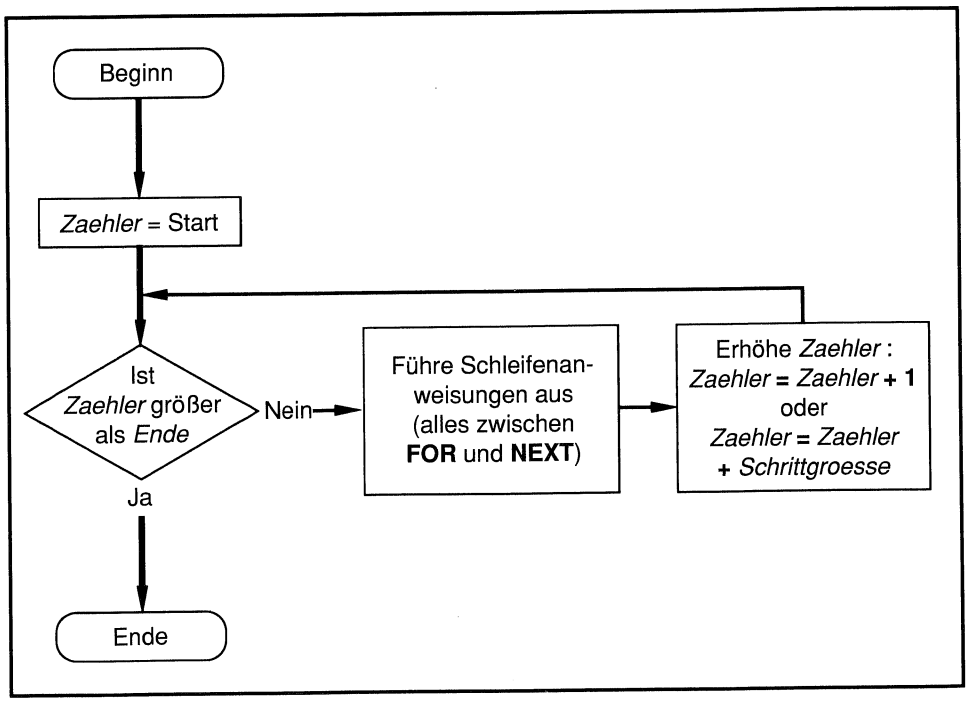
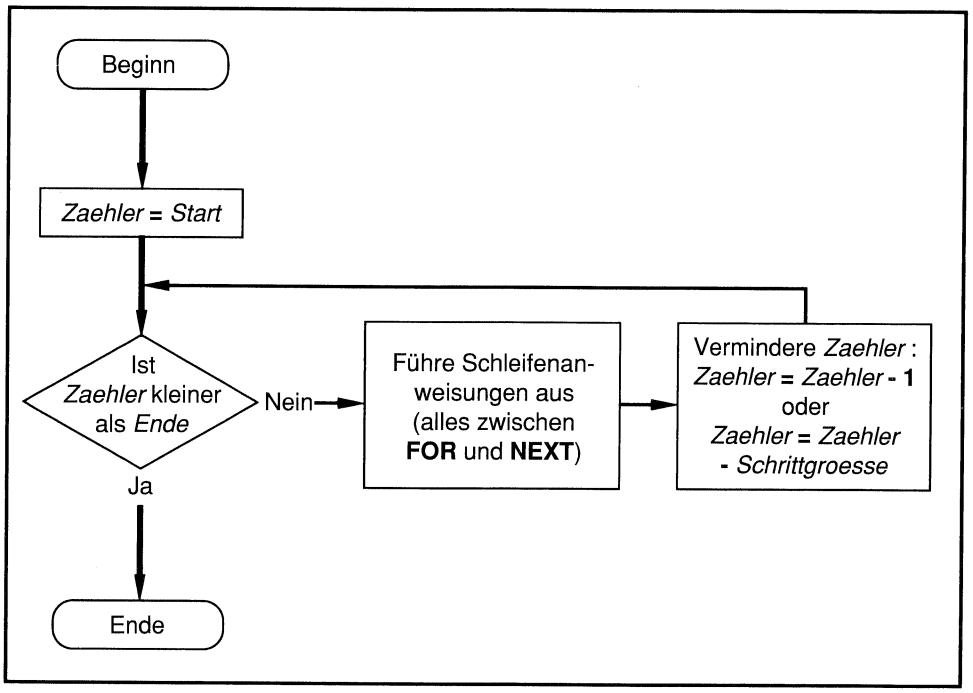


Abbildung 1.2 zeigt die Logik einer **FOR...NEXT**-Schleife, wenn der Wert von *Schrittgröße* negativ ist.

*Abbildung 1.2 Logik einer **FOR...NEXT**-Schleife mit negativer **STEP**-Größe*




Eine **FOR...NEXT**-Anweisung überprüft immer am Kopf der Schleife, so daß die Schleife nie ausgeführt wird, wenn eine der folgenden Bedingungen wahr ist:

- Die Schrittgröße ist positiv, und der Anfangswert von *Start* ist größer als der Wert von *Ende*:

```
' Schleife wird nie ausgeführt, da der Anfangswert von
' I% größer als 9 ist:
FOR I% = 10 TO 9
.
.
.
NEXT I%
```

- Die Schrittgröße ist negativ, und der ursprüngliche Wert von *Start* ist kleiner als der Wert von *Ende*:

```
' Schleife wird nie ausgeführt, da I% bereits zu
' Beginn kleiner als -9 ist:
FOR I% = -10 TO -9 STEP -1
.
.
.
NEXT I%
```

In der **NEXT**-Klausel müssen Sie das Argument *Zähler* nicht verwenden. Wenn Sie jedoch mehrere verschachtelte **FOR...NEXT**-Schleifen (eine Schleife innerhalb einer anderen) haben, kann das Aufführen der Argumente *Zähler* hilfreich sein, um zu erkennen, in welcher Schleife Sie sich befinden.

Hier sind einige generelle Regeln zur Verschachtelung von **FOR...NEXT**-Schleifen:

- Wenn Sie eine Schleifenzählervariable in einer **NEXT**-Klausel verwenden, muß der Zähler einer eingeschlossenen Schleife vor dem Zähler der umschließenden Schleife stehen. Mit anderen Worten, die folgende Verschachtelung ist zulässig:

```
FOR I = 1 TO 10
   FOR J = -5 TO 0
   .
   .
   .
   NEXT J
NEXT I
```

Die folgende Verschachtelung ist jedoch nicht zulässig:

```
FOR I = 1 TO 10
   FOR J = -5 TO 0
   .
   .
   .
   NEXT I
NEXT J
```

- Wenn Sie eine separate **NEXT**-Klausel zum Beenden jeder Schleife benutzen, muß die Anzahl der **NEXT**-Klauseln mit der Anzahl der **FOR**-Klauseln übereinstimmen.

- Wenn Sie eine einzelne **NEXT**-Klausel zum Beenden verschiedener Ebenen von **FOR...NEXT**-Schleifen verwenden, müssen die Schleifenzählervariablen nach der **NEXT**-Klausel in einer "von innen nach außen"-Reihenfolge erscheinen:

  **NEXT** *innerster Schleifenzähler,...,äußerster Schleifenzähler*

  In diesem Fall muß die Anzahl der Schleifenzählervariablen in der **NEXT**-Klausel der Anzahl der **FOR**-Klauseln entsprechen.

### Beispiele

Die folgenden drei Programmausschnitte zeigen verschiedene Möglichkeiten des Verschachtelns von **FOR...NEXT**-Schleifen, um die gleiche Ausgabe zu erzeugen.

Das nächste Beispiel zeigt verschachtelte **FOR...NEXT**-Schleifen mit Schleifenzählern und separaten **NEXT**-Klauseln für jede Schleife:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
      NEXT K
   NEXT J
NEXT I
```

Das folgende Beispiel benutzt ebenfalls Schleifenzähler, jedoch nur eine **NEXT**-Klausel für alle drei Schleifen:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
NEXT K, J, I
```

Das letzte Beispiel zeigt verschachtelte **FOR...NEXT**-Schleifen ohne Schleifenzähler:

```
FOR I = 1 TO 2
   FOR J = 4 TO 5
      FOR K = 7 TO 8
         PRINT I, J, K
      NEXT
   NEXT
NEXT
```

***Ausgabe***

```
 1            4            7
 1            4            8
 1            5            7
 1            5            8
 2            4            7
 2            4            8
 2            5            7
 2            5            8
```

### 1.4.1.1 Das Verlassen einer FOR...NEXT-Schleife mit EXIT FOR

Manchmal möchten Sie vielleicht eine **FOR...NEXT**-Schleife verlassen, bevor die Zählervariable den Endwert der Schleife erreicht hat. Dies können Sie mit der **EXIT FOR**-Anweisung tun. Eine einzige **FOR...NEXT**-Schleife kann eine beliebige Anzahl von **EXIT FOR**-Anweisungen enthalten, wobei die **EXIT FOR**-Anweisungen an beliebiger Stelle innerhalb der Schleife stehen können. Der folgende Ausschnitt zeigt eine Verwendung einer **EXIT FOR**-Anweisung:

```
' Schreibe die Quadratwurzeln der Zahlen von 1 bis
' 30.000.
' Wenn der Benutzer eine Taste während der
' Schleifenausführung betätigt, verläßt die Kontrolle
' die Schleife:
FOR I% = 1 TO 30000
   PRINT SQR(I%)
   IF INKEY$ <> "" THEN EXIT FOR
NEXT
.
.
.
```

**EXIT FOR** verläßt nur die innerste eingeschlossene **FOR...NEXT**-Schleife, in der es vorkommt. Wenn beispielsweise der Benutzer während der Ausführung der folgenden verschachtelten Schleifen eine Taste betätigt, würde das Programm nur die innere Schleife verlassen. Wenn die äußere Schleife immer noch aktiv wäre (das heißt, der Wert von `I%` ist <= 100), würde die Steuerung direkt an die innere Schleife zurückgegeben:

```
FOR I% = 1 TO 100
   FOR J% = 1 TO 100
      PRINT I%/J%
      IF INKEY$ <> "" THEN EXIT FOR
   NEXT J%
 NEXT I%
```

#### 1.4.1.2 Verzögerung der Programmausführung mit einer FOR...NEXT-Schleife

Viele BASICA-Programme verwenden eine leere **FOR...NEXT**-Schleife wie die folgende, um in einem Programm eine Verzögerung einzufügen:

```
.
.
.
' Es befinden sich keine Anweisungen in dem Körper
' dieser Schleife; alles, was sie tut, ist von 1 bis
' 10.000 unter Verwendung von Ganzzahlen zu zählen.
FOR I% = 1 TO 10000 : NEXT
.
.
.
```

Für sehr kurze Verzögerungen oder für Verzögerungen, die kein exaktes Intervall sein müssen, eignet sich die Verwendung von **FOR...NEXT** hervorragend. Das Problem beim Verwenden leerer **FOR...NEXT**-Schleifen in dieser Art und Weise ist, daß sowohl verschiedene Computer als auch verschiedene BASIC-Versionen sowie verschiedene Kompilier-Optionen Auswirkung darauf haben können, wie schnell die Berechnungen in einer **FOR...NEXT**-Schleife ausgeführt werden. Daraus folgt, daß die Länge einer Verzögerung sehr unterschiedlich sein kann. Abschnitt 1.4.3.3 , "Verzögerung der Programmausführung mit **DO...LOOP**", zeigt eine bessere Alternative.

### 1.4.2 WHILE...WEND-Schleifen

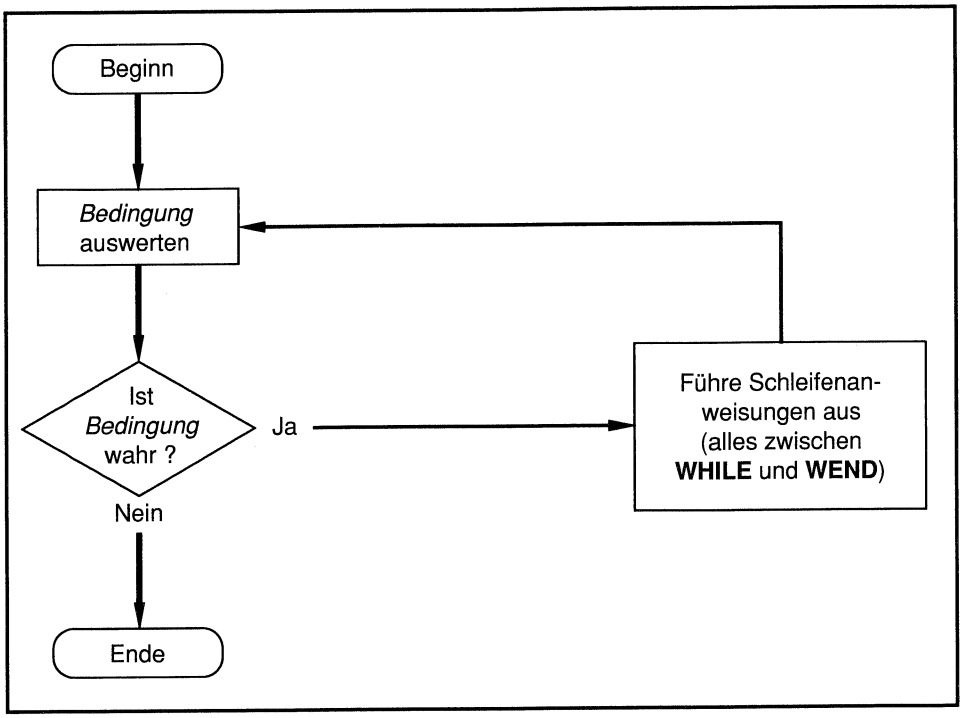
Die **FOR...NEXT**-Anweisung ist vorteilhaft, wenn Sie bereits vorher wissen, wie oft eine Schleife ausgeführt werden soll. Wenn Sie jedoch die genaue Anzahl, wie oft eine Schleife ausgeführt werden soll, nicht vorhersehen können, aber die Bedingung kennen, die die Schleife abbricht, ist die **WHILE...WEND**-Anweisung vorteilhaft. Anstatt per Zählung festzustellen, ob eine Schleife weiterhin ausgeführt werden soll, wiederholt **WHILE...WEND** die Schleife solange die Schleifenbedingung wahr ist.

Die folgende Tabelle zeigt die Syntax der **WHILE...WEND**-Anweisung und ein Beispiel ihrer Anwendung.

| *Syntax* | *Beispiel* |
|---|---|
| **WHILE** *Bedingung*<br>*[Anweisungsblock]*<br>**WEND** | `INPUT A$`<br>`WHILE A$ <> "J" AND A$ <> "N"`<br>`  PRINT "Antwort muß J oder N sein."`<br>`  INPUT A$`<br>`WEND` |

Abbildung 1.3 zeigt die Logik einer **WHILE...WEND**-Schleife.

*Abbildung 1.3 Logik einer **WHILE...WEND**-Schleife*




### Beispiel

Das folgende Beispiel weist der Variablen X einen Anfangswert von zehn zu und halbiert den Wert solange, bis der Wert von X kleiner als 0,00001 ist:

```
 X = 10
WHILE X > 0.00001
   PRINT X
   X = X/2
WEND
```

## 1.4.3 DO...LOOP-Schleifen

Wie die **WHILE...WEND**-Anweisung führt die **DO...LOOP**-Anweisung einen Block von Anweisungen unbestimmt oft aus; das bedeutet, daß das Verlassen der Schleife davon abhängt, ob die Schleifenbedingung falsch oder wahr ist. Anders als **WHILE...WEND** erlaubt **DO...LOOP** Ihnen, entweder auf eine wahre oder auf eine falsche Bedingung hin zu prüfen. Zudem kann die Prüfung der Bedingung sowohl am Beginn als auch am Ende der Schleife stehen.

Die folgende Tabelle zeigt die Syntax einer Schleife, die die Prüfung am Anfang vornimmt.

| *Syntax* | *Beispiel* |
|---|---|
| **DO** [{**WHILE** \| **UNTIL**} *Bedingung*]<br>[*Anweisungsblock*]<br>[**EXIT DO**]<br>**LOOP** | `DO UNTIL INKEY$ <> ""`<br>`  ' Ein leere Schleife die`<br>`  ' verzögert bis eine Taste`<br>`  ' betätigt wird`<br>`LOOP` |

Die Abbildungen 1.4 und 1.5 zeigen die beiden Arten von **DO...LOOP**, die die Prüfung am Anfang der Schleife vornehmen.

*Abbildung 1.4 Die Logik von **DO WHILE...LOOP***

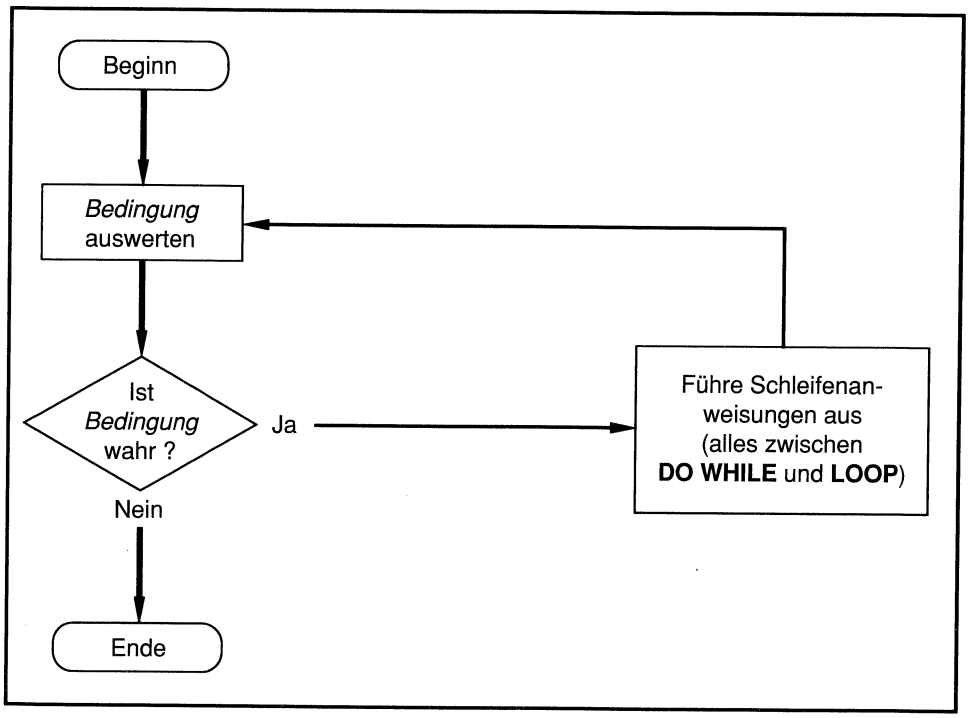
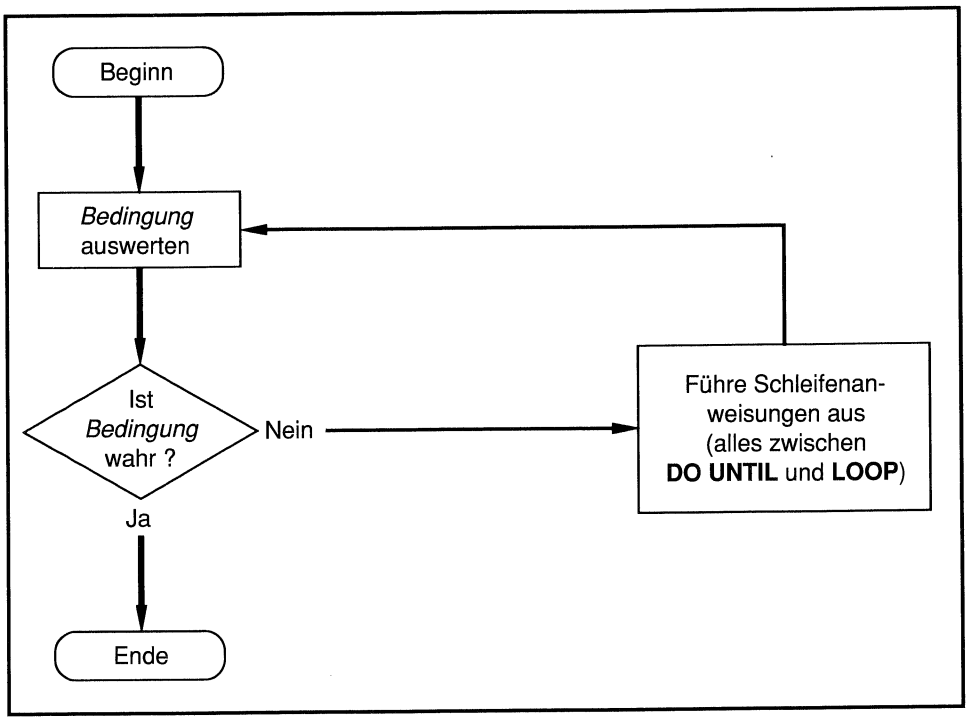


*Abbildung 1.5 Die Logik von **DO UNTIL...LOOP***




Die folgende Tabelle zeigt die Syntax einer Schleife, die am Ende der Schleife auf wahr oder falsch prüft.

| *Syntax* | *Beispiel* |
|---|---|
| **DO**<br>[*Anweisungsblock*]<br>[**EXIT DO**]<br>**LOOP** [{**WHILE** \| **UNTIL**} *Bedingung*] | `DO`<br>`  INPUT "Verändern: ", Ve`<br>`  Gesamt = Gesamt + Ve`<br>`LOOP WHILE Ve <> 0` |

Abbildungen 1.6 und 1.7 zeigen die beiden Arten der **DO...LOOP**, die am Ende der Schleife prüfen.

*Abbildung 1.6 Die Logik von **DO...LOOP WHILE***

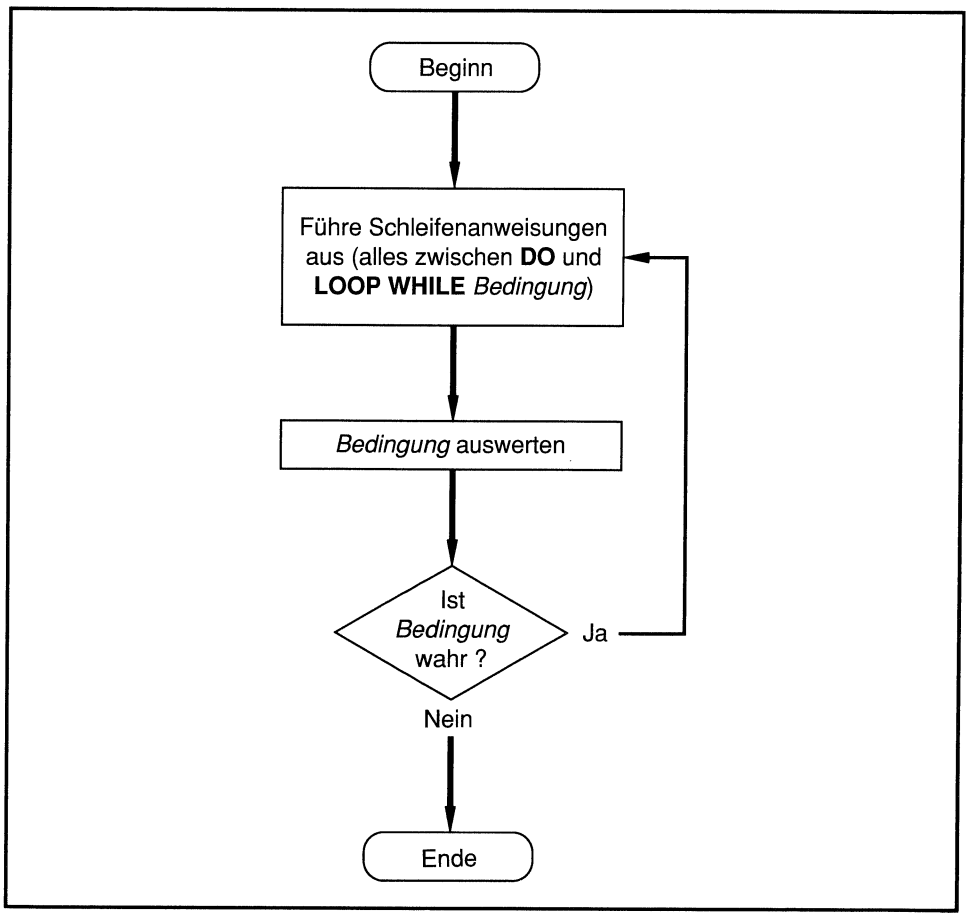
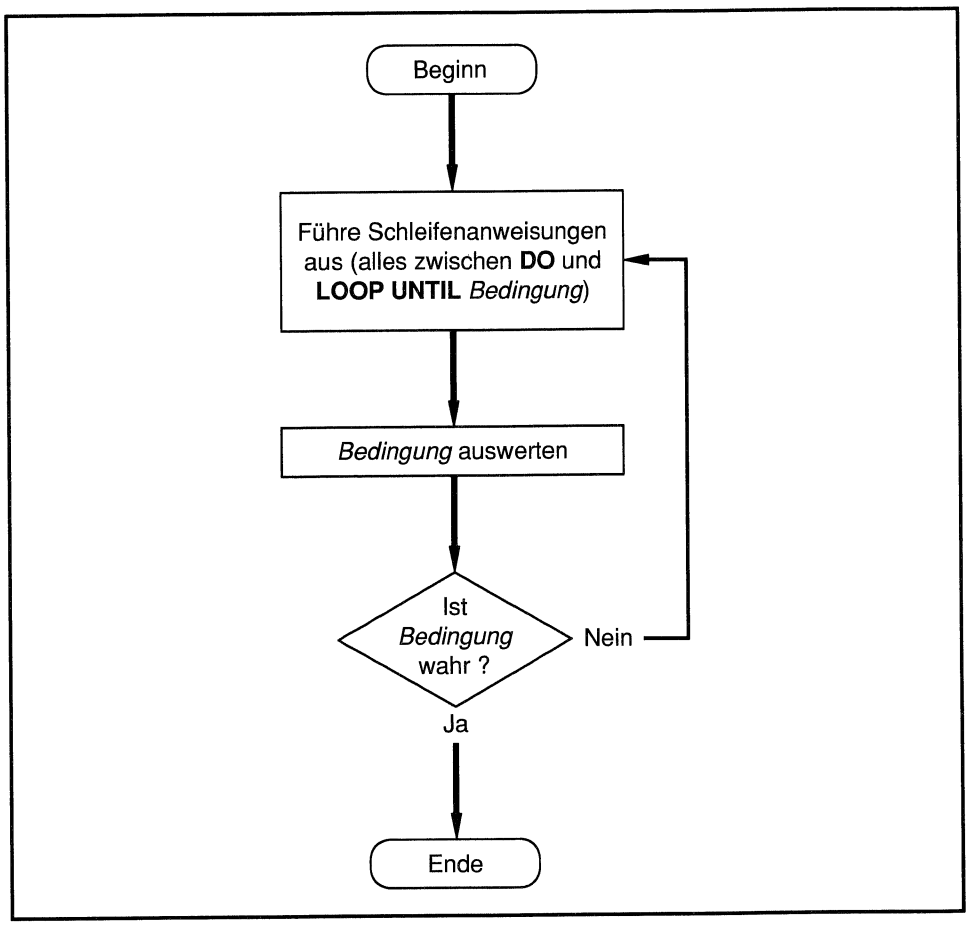


*Abbildung 1.7 Die Logik von **DO...LOOP UNTIL***




### 1.4.3.1 Schleifenprüfungen: Eine Möglichkeit, DO...LOOP zu verlassen

Wenn Sie die Schleifenprüfung an das Ende einer **DO...LOOP**-Anweisung setzen, haben Sie eine Schleife, in der die Anweisungen mindestens einmal ausgeführt werden. Bei der **WHILE...WEND**-Anweisung müssen Sie manchmal auf einen Programmiertrick zurückgreifen, um den ersten Schleifendurchlauf zu erzwingen. Mit **DO...LOOP** sind solche Tricks nicht mehr notwendig, wie unten gezeigt:

```
' WHILE...WEND-Schleife prüft am Kopf, so daß die
' Zuweisung von "J" an Antwort$ notwendig ist, um
' mindestens eine Ausführung der Schleife zu
' erzwingen:
```

```
Antwort$ = "J"
WHILE UCASE$(Antwort$) = "J"
    .
    .
    .
    INPUT "Noch einmal"; Antwort$
WEND
' Dieselbe Schleife mit DO...LOOP,
' sie überprüft nach dem Schleifenkörper:
DO
   .
   .
   .
   INPUT "Noch einmal"; Antwort$
LOOP WHILE UCASE$(Antwort$) = "J"
```

Sie können eine mit **WHILE** formulierte Bedingung immer neu schreiben, indem Sie stattdessen **UNTIL** verwenden, wie im Folgenden gezeigt:

```
' ==========================================
'    Verwendung von DO WHILE NOT...LOOP
' ==========================================
'
'Solange das Ende der Datei 1 nicht erreicht ist,
'lies eine Zeile aus der Datei und schreibe sie auf
'den Bildschirm:
DO WHILE NOT EOF (1)
   LINE INPUT #1, ZeilenPuffer$
   PRINT ZeilenPuffer$
LOOP
' ==========================================
'      Verwendung von DO UNTIL...LOOP
' ==========================================
'
' Bis das Ende von Datei 1 erreicht ist , lies eine
' Zeile aus der Datei und gib sie auf den Bildschirm
' aus:
DO UNTIL EOF(1)
   LINE INPUT #1, ZeilenPuffer$
   PRINT ZeilenPuffer$
LOOP
```

### 1.4.3.2 EXIT DO: Ein weiterer Weg, DO...LOOP zu verlassen

Sehr wahrscheinlich passiert innerhalb des Körpers einer **DO...LOOP**-Anweisung etwas, das letztlich die Schleifenabbruchbedingung von wahr auf falsch oder von falsch auf wahr verändert, so daß die Schleife beendet wird. In den bisher aufgeführten Beispielen zu **DO...LOOP** erschien die Abbruchbedingung entweder am Beginn oder Ende der Schleife. Wenn Sie jedoch für das Verlassen der Schleife die Anweisung **EXIT DO** verwenden, können Sie die Abbruchbedingung innerhalb der Schleife plazieren. Eine einzelne **DO...LOOP**-Schleife kann eine beliebige Anzahl von **EXIT DO**-Anweisungen enthalten, und die **EXIT DO**-Anweisungen können an beliebiger Stelle innerhalb der Schleife erscheinen.

#### Beispiel

Im folgenden Beispiel wird eine Datei geöffnet und Zeile für Zeile eingelesen, bis das Dateiende erreicht ist, oder das vom Benutzer eingegebene Muster entdeckt wurde. Wenn das Muster entdeckt wird bevor das Dateiende erreicht ist, wird die Schleife mit einer **EXIT DO**-Anweisung verlassen.

```
INPUT "Zu durchsuchende Datei: ", Datei$
IF Datei$ = "" THEN END
INPUT "Zu suchendes Muster: ", Muster$
OPEN Datei$ FOR INPUT AS #1
DO UNTIL EOF(1)     ' EOF(1) gibt einen wahren Wert an,
                    ' wenn das Dateiende erreicht ist.
   LINE INPUT #1, TempZeile$
   IF INSTR(TempZeile$, Muster$) > 0 THEN
      ' Gib die erste Zeile aus, die das Muster
      ' enthält, und verlasse die Schleife:
      PRINT TempZeile$
      EXIT DO
   END IF
LOOP
```

### 1.4.3.3 Verzögerung der Programmausführung mit DO...LOOP

Wenn es wichtig ist, daß eine Verzögerung in Ihrem Programm immer die gleiche Länge hat, sollten Sie anstelle einer **FOR...NEXT**-Schleife eine **DO...LOOP**-Schleife verwenden. Die Verwendung eines **DO...LOOP** als Verzögerungsschleife macht Ihr Programm unabhängig von arithmetischen Berechnungsgeschwindigkeiten.

**Beispiel**

Der folgende Ausschnitt enthält eine **SUB**-Prozedur, die für die Anzahl von Sekunden verzögert, die der Prozedur übergeben wird.

```
.
.
.
CALL VerzSchleife(15.5)   ' Verzögere für 15,5 Sekunden
.
.
.
END
SUB VerzSchleife(VerzZeit) STATIC
    CONST SekAmTag = 24& * 60& * 60&
    ' TIMER ist eine BASIC-Funktion, die die Anzahl
    ' der seit 0.00 Uhr vergangenen Sekunden angibt. Das
    ' Unterprogramm VerzSchleife wird verlassen, wenn
    ' TIMER größer als SchleifEnde ist.
    SchleifEnde = TIMER + VerzZeit
    ' Wenn die Verzögerung vor 0.00 Uhr beginnt und
    ' bis nach 0.00 Uhr dauert:
    IF SchleifEnde > SekAmTag THEN
       SchleifEnde = SchleifEnde - SekAmTag
       ' Warte, bis die Uhr 0.00 Uhr überschritten
       ' hat und auf null zurückgesetzt wird:
       DO WHILE TIMER > SchleifEnde
       LOOP
    END IF
    ' Verzögere solange die Uhr kleiner als die
    ' Beendigungszeit ist:
    DO WHILE TIMER < SchleifEnde
    LOOP
END SUB
```

## 1.5 Anwendungsbeispiele

Die Anwendungsbeispiele in diesem Kapitel sind ein Programm für eine Scheckheftbilanz und ein Programm, das sicherstellt, daß jede Zeile in einer Textdatei mit einer Sequenz Wagenrücklauf-Zeilenvorschub endet.

## 1.5.1 Programm zur Scheckheftbilanz (SCHECK.BAS)

Dieses Programm fordert den Benutzer auf, die Anfangsbilanz des Scheckkontos und alle Bewegungen – Entnahmen und Einlagen – die sich ereignet haben, einzugeben. Das Programm gibt dann eine sortierte Liste der Bewegungen und die Schlußbilanz des Kontos aus.

### Verwendete Anweisungen und Funktionen

Dieses Programm zeigt die folgenden in diesem Kapitel erläuterten Anweisungen:

- **DO...LOOP WHILE**
- **FOR...NEXT**
- **EXIT FOR**
- Block-**IF...THEN...ELSE**

### Programm-Listing

```
DIM Betrag(1 TO 100)
CONST FALSCH = 0, WAHR = NOT FALSCH
' Hole die Anfangsbilanz des Kontos:
CLS
PRINT "Geben Sie die Anfangsbilanz ein, und betätigen Sie";
PRINT " die Eingabetaste";
INPUT ":", Bilanz
' Hole die Transaktionen. Fahre mit der Annahme der
' Transaktionen fort, bis null für eine Transaktion
' eingegeben wird, oder bis 100 Transaktionen eingegeben
' sind:
FOR TransNum% = 1 TO 100
   PRINT TransNum%;
   PRINT ") Eingabe des Bewegungsbetrages "
   PRINT "(0 zum beenden): ";
   INPUT "", Betrag(TransNum%)
   IF Betrag(TransNum%) = 0 THEN
      TransNum% = TransNum% - 1
      EXIT FOR
   END IF
NEXT
' Sortiere Transaktionen in aufsteigender Reihenfolge
' unter Verwendung eines "Bubblesort":
Limit% = TransNum%
```

```
DO
   Tausch% = FALSCH
   FOR I% = 1 TO (Limit%-1)
      ' Wenn zwei aufeinanderfolgende Elemente nicht
      ' in der richtigen Reihenfolge sind, tausche
      ' diese Elemente:
      IF Betrag(I%) < Betrag(I%+1) THEN
         SWAP Betrag(I%), Betrag(I%+1)
         Tausch% = I%
      END IF
   NEXT I%
   ' Sortiere im nächsten Schritt nur bis dahin, wo
   ' der letzte Tausch vorgenommen wurde:
   IF Tausch% THEN Limit% = Tausch%
' Sortiere, bis keine Elemente mehr getauscht sind:
LOOP WHILE Tausch%
' Gib das sortierte Datenfeld der Transaktionen aus. Wenn
' eine Bewegung größer als null ist, gib sie als
' "EINZAHLUNG" aus, wenn sie kleiner als null ist, gib
' sie als "AUSZAHLUNG" aus:
FOR I% = 1 TO TransNum%
   IF Betrag(I%) > 0 THEN
      PRINT USING "EINZAHLUNG: DM#####,##"; Betrag(I%)
   ELSEIF Betrag(I%) < 0 THEN
      PRINT USING "AUSZAHLUNG: DM#####,##"; Betrag(I%)
   END IF
   ' Aktualisiere Bilanz:
   Bilanz = Bilanz + Betrag(I%)
NEXT I%
' Gib die Schlußbilanz aus:
PRINT
PRINT "-------------------------"
PRINT USING "Endergebnis: DM######,##"; Bilanz
END
```

## 1.5.2 Filter für Wagenrücklauf-Zeilenvorschub (WRZV.BAS)

Einige Textdateien sind in einem Format gespeichert, das nur einen Wagenrücklauf (Rücklauf zum Beginn der Zeile) oder einen Zeilenvorschub (Vorschub auf die nächste Zeile) verwendet, um das Ende einer Zeile anzuzeigen. Viele Texteditoren erweitern diesen einen Wagenrücklauf (WR) oder Zeilenvorschub (ZV) in eine Sequenz Wagenrücklauf-Zeilenvorschub (WR-ZV) immer dann, wenn Sie die Datei zum Editieren laden.

Wenn Sie jedoch einen Texteditor verwenden, der ein einzelnes WR oder ZV nicht in ein WR-ZV erweitert, müssen Sie die Datei vielleicht so verändern, daß sie am Ende jeder Zeile die richtige Sequenz hat. Das folgende Programm ist ein Filter, der die Datei öffnet, ein einzelnes WR oder ZV in eine Kombination WR-ZV erweitert und anschließend die ausgerichtete Zeile in eine neue Datei schreibt. Der ursprüngliche Inhalt der Datei wird in einer Datei mit der Namenserweiterung `.BAK` gespeichert.

### Verwendete Anweisungen und Funktionen

Dieses Programm zeigt die folgenden in diesem Kapitel erläuterten Anweisungen:

- **DO...LOOP WHILE**
- **DO UNTIL...LOOP**
- Block-**IF...THEN...ELSE**
- **SELECT CASE...END SELECT**

Um dieses Programm nützlicher für Sie zu gestalten, enthält es ebenfalls die folgenden Konstruktionen, die in diesem Kapitel nicht erläutert sind:

- Eine **FUNCTION**-Prozedur mit dem Namen `Sicherung$`, die eine Datei mit der Erweiterung `.BAK` anlegt.

  In Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", finden Sie weitere Informationen zur Definition und Verwendung von Prozeduren.
- Eine Fehlerbehandlungsroutine mit dem Namen `FehlerBehand`, um Fehler zu behandeln, die auftreten können, wenn der Benutzer einen Dateinamen eingibt. Wenn der Benutzer beispielsweise den Namen einer nicht existierenden Datei eingegeben hat, fragt die Routine erneut nach einem Namen. Ohne diese Routine würde ein solcher Fehler das Programm beenden.

  Weitere Informationen zur Verfolgung von Fehlern finden Sie in Kapitel 6, "Fehler- und Ereignisverfolgung".

### Programm-Listing

```
DEFINT A-Z          'Standardvariablentyp ist Ganzzahl.
' Die FUNCTION Sicherung$ erstellt eine
' Sicherungsdatei mit dem Basisnamen der
' der zu verändernden Datei plus einer
' der Namenserweiterung .BAK:
DECLARE FUNCTION Sicherung$ (DateiName$)
' Initialisiere symbolische Konstanten und Variablen:
CONST FALSCH = 0, WAHR = NOT FALSCH
```

```
DruRueck$ = CHR$(13)
ZeilenVor$ = CHR$(10)
DO
   CLS
   ' Eingabe des Namens der zu verändernden Datei:
   INPUT "Welche Datei wollen Sie übertragen"; AusDatei$
   InDatei$ = Sicherung$(AusDatei$) ' Hole den Namen
                                    ' der
                                    ' Sicherungsdatei.
   ON ERROR GOTO FehlerBehand  ' Schalte
                               ' Fehlerverfolgung ein.
   NAME AusDatei$ AS InDatei$  ' Kopiere die
                               ' Eingabedatei in die
                               ' Sicherungsdatei.
   ON ERROR GOTO 0             ' Schalte die
                               ' Fehlerverfolgung aus.
   ' Öffne die Sicherungsdatei für Eingabe und die
   ' alte Datei für Ausgabe:
   OPEN InDatei$ FOR INPUT AS #1
   OPEN AusDatei$ FOR OUTPUT AS #2
   ' Die Variable VorDruRueck ist eine Flagge, die
   ' immer dann auf wahr gesetzt wird, wenn das
   ' Programm ein Wagenrücklaufzeichen liest:
   VorDruRueck = FALSCH
   ' Lies aus der Eingabedatei bis das Dateiende
   ' erreicht ist:
   DO UNTIL EOF(1)
      ' Dies ist nicht das Ende der Datei, lies
      ' deshalb ein Zeichen:
      DateiZeich$ = INPUT$(1, #1)
      SELECT CASE DateiZeich$
          CASE DruRueck$     ' Das Zeichen ist ein WR.
             ' Wenn das vorhergehende Zeichen
             ' ebenfalls ein WR war, schreibe ein ZV
             ' vor das Zeichen:
             IF VorDruRueck THEN
                DateiZeich$ = ZeilenVor$ + DateiZeich$
             END IF
             ' Setze in jedem Fall die Variable
             ' VorDruRueck auf WAHR:
             VorDruRueck = WAHR
```

```
            CASE ZeilenVor$      ' Das Zeichen ist ein ZV.
               ' Wenn das vorhergehende Zeichen kein WR
               ' war, schreibe ein WR vor das Zeichen:
               IF NOT VorDruRueck THEN
                  DateiZeich$ = DruRueck$ + DateiZeich$
              END IF
              ' Setze die Variable VorDruRueck in jedem
              ' Fall auf FALSCH:
              VorDruRueck = FALSCH
            CASE ELSE          ' Weder ein WR noch ein ZV.
               ' Wenn das vorhergehende Zeichen ein WR
               ' war, dann setze die Variable VorDruRueck
               ' auf FALSCH und schreibe ein ZV vor das
               ' aktuelle Zeichen:
              IF VorDruRueck THEN
                 VorDruRueck = FALSCH
                 DateiZeich$ = ZeilenVor$ + DateiZeich$
              END IF
         END SELECT
         ' Schreibe das/die Zeichen in die neue Datei:
         PRINT #2, DateiZeich$;
      LOOP
      ' Schreibe ein ZV, wenn das letzte Zeichen der
      ' Datei ein WR war:
      IF VorDruRueck THEN PRINT #2, ZeilenVor$;
      CLOSE                      ' Schließe beide Dateien.
      PRINT "Eine weitere Datei (J/N)?"  ' Anfrage zum
                                         ' Fortführen.
      ' Verändere die Eingabe in Großbuchstaben:
      Weiter$ = UCASE$(INPUT$(1))
   ' Setze das Programm fort, wenn der Benutzer ein "j"
   ' oder "J" eingegeben hat:
   LOOP WHILE Weiter$ = "J"
   END
   FehlerBehand:           ' Fehlerbehandlungsroutine
      CONST KEINEDATEI = 53, DATEIDA = 58
      ' Die Funktion ERR gibt den Fehlercode des letzten
      ' Fehlers an:
```

```
   SELECT CASE ERR
      CASE KEINEDATEI  ' Programm konnte Datei mit dem
                       ' eingegebenen Namen nicht
                       ' finden.
         PRINT "Keine solche Datei im aktuellen "
         PRINT "Verzeichnis vorhanden."
         INPUT "Geben Sie neuen Namen ein: ",AusDatei$
         InDatei$ = Sicherung$(AusDatei$)
         RESUME
      CASE DATEIDA     ' Der Dateiname <Dateiname>.BAK
                       ' existiert bereits in diesem
                       ' Verzeichnis: entferne die
                       ' Datei und fahre dann fort.
         KILL InDatei$
         RESUME
      CASE ELSE        ' Ein unerwarteter Fehler trat
                       ' auf: beende das Programm.
         ON ERROR GOTO 0
   END SELECT
'
' ================== SICHERUNG$ ====================
'   Diese Prozedur gibt einen Dateinamen an, der aus
'   dem Basisnamen der eingegebenen Datei (alles vor
'   dem ".") plus der Erweiterung ".BAK" besteht.
' ==================================================
'
FUNCTION Sicherung$ (DateiName$) STATIC
   ' Suche nach einem Punkt:
   Erweiterung = INSTR(DateiName$, ".")
   ' Wenn ein Punkt vorhanden ist, füge .BAK zu der
   ' Basis hinzu:
   IF Erweiterung > 0 THEN
      Sicherung$ = LEFT$(DateiName$, Erweiterung-1) + ".BAK"
   ' Andernfalls füge dem ganzen Namen .BAK hinzu:
   ELSE
      Sicherung$ = DateiName$ + ".BAK"
   END IF
END FUNCTION
```

## 1.6 Zusammenfassung der Anweisungen zur Ablaufsteuerung

Die folgende Tabelle führt die in BASIC zur Programmablaufsteuerung verwendeten Anweisungen auf.

| *Aufgabe* | *Anweisung* | *Funktion* |
|---|---|---|
| Schleife | **FOR...NEXT** | Wiederholt Anweisungen zwischen **FOR** und **NEXT** sooft wie angegeben |
| | **EXIT FOR** | Bietet einen weiteren Weg zum Verlassen einer **FOR...NEXT**-Schleife |
| | **DO LOOP** | Wiederholt Anweisungen zwischen **DO** und **LOOP** entweder bis eine angegebene Bedingung wahr ist (**DO...LOOP UNTIL** *Bedingung*) oder solange eine gegebene Bedingung wahr ist (**DO...LOOP WHILE** *Bedingung*) |
| | **EXIT DO** | Bietet einen weiteren Weg, eine **DO...LOOP**-Schleife zu verlassen |
| | **WHILE...WEND** | Wiederholt Anweisungen zwischen **WHILE** und **WEND** solange eine gegebene Bedingung wahr ist (ähnlich wie **DO WHILE** *Bedingung*...**LOOP**) |
| Entscheidungsfindung | **IF...THEN...ELSE** | Führt verschiedene Anweisungen aus oder verzweigt zu verschiedenen Anweisungen, abhängig von gewissen Bedingungen. |
| | **SELECT CASE** | Führt verschiedene Anweisungen, abhängig von gewissen Bedingungen aus. |

# 2 Prozeduren: Unterprogramme und Funktionen


2.1 Prozeduren: Bausteine für die Programmierung 2.3

2.2 Prozeduren im Vergleich mit Unterroutinen und Funktionen 2.4
- 2.2.1 **SUB** und **GOSUB** im Vergleich 2.5
- 2.2.2 **FUNCTION** im Vergleich mit **DEF FN** 2.74

2.3 Wie Sie Prozeduren definieren 2.9

2.4 Wie Sie Prozeduren aufrufen 2.11
- 2.4.1 Wie Sie eine **FUNCTION**-Prozedur aufrufen 2.11
- 2.4.2 Wie Sie eine **SUB**-Prozedur aufrufen 2.12

2.5 Wie Sie Argumente an Prozeduren übergeben 2.14
- 2.5.1 Parameter und Argumente 2.14
- 2.5.2 Wie Sie Konstanten und Ausdrücke übergeben 2.16
- 2.5.3 Wie Sie Variablen übergeben 2.17
  - 2.5.3.1 Wie Sie einfache Variablen übergeben 2.18
  - 2.5.3.2 Wie Sie ein vollständiges Datenfeld übergeben 2.19
  - 2.5.3.3 Wie Sie einzelne Datenfeldelemente übergeben 2.19
  - 2.5.3.4 Wie Sie die Funktionen für Datenfeldgrenzen verwenden 2.20
  - 2.5.3.5 Wie Sie einen vollständigen Verbund übergeben 2.21
  - 2.5.3.6 Wie Sie einzelne Elemente eines Verbundes übergeben 2.22
- 2.5.4 Überprüfen von Argumenten mit der Anweisung **DECLARE** 2.22
  - 2.5.4.1 Wann QuickBASIC keine **DECLARE**-Anweisung erzeugt 2.23
  - 2.5.4.2 Wie Sie Programme außerhalb der QuickBASIC-Umgebung entwickeln 2.24
  - 2.5.4.3 Wie Sie Include-Dateien für Deklarationen verwenden 2.26
  - 2.5.4.4 Wie Sie Prozeduren in Quick-Bibliotheken deklarieren 2.29
- 2.5.5 Wie Sie Argumente nach Referenz übergeben 2.29
- 2.5.6 Wie Sie Argumente nach Wert übergeben 2.30

2.6 Die gemeinsame Nutzung von Variablen mit **SHARED** 2.32
- 2.6.1 Die gemeinsame Nutzung von Variablen mit bestimmten Prozeduren in einem Modul 2.32
- 2.6.2 Die gemeinsame Nutzung von Variablen mit allen Prozeduren in einem Modul 2.34
- 2.6.3 Die gemeinsame Nutzung von Variablen mit anderen Modulen 2.37
- 2.6.4 Das Problem von adreßgleichen Variablen 2.39

2.7 Automatische und **STATIC**-Variablen 2.40

2.8 Beibehaltung der Werte von lokalen Variablen mit der **STATIC**-Anweisung 2.41

2.9 Rekursive Prozeduren 2.42
- 2.9.1 Die Funktion Fakultät 2.43
- 2.9.2 Anpassen der Stapelgröße 2.44

2.10 Übertragen der Kontrolle an ein anderes Programm mit CHAIN 2.45

2.11 Beispielanwendung: Rekursive Verzeichnisdurchsuchung (WOIST.BAS) 2.47

2.12 Zusammenfassung der Anweisungen, die in BASIC-Prozeduren verwendet werden 2.52

Dieses Kapitel beschreibt, wie Programme durch Aufteilen in kleinere logische Komponenten vereinfacht werden. Diese Komponenten – Prozeduren genannt – können dann zu Bausteinen werden, mit denen Sie die Sprache BASIC selbst verbessern und erweitern können.

Nach dem Lesen dieses Kapitels werden Sie wissen, wie Sie die folgenden Aufgaben mit Prozeduren bewerkstelligen:

- Definition und Aufruf von BASIC-Prozeduren.
- Verwenden von globalen und lokalen Variablen in Prozeduren.
- Verwenden von Prozeduren anstelle von **GOSUB**-Unterroutinen und **DEF FN**-Funktionen.
- Übergabe von Argumenten an Prozeduren und Rückgabe von Werten aus Prozeduren.
- Schreiben rekursiver Prozeduren (Prozeduren, die sich selbst aufrufen können).

***Hinweis*** Obwohl Sie ein BASIC-Programm mit jedem beliebigen Texteditor erstellen können, macht es Ihnen der QuickBASIC-Editor sehr leicht, Programme mit Prozeduren zu schreiben. Zudem erzeugt QuickBASIC in den meisten Fällen automatisch eine **DECLARE**-Anweisung, wenn Sie Ihr Programm speichern. (Eine **DECLARE**-Anweisung stellt sicher, daß nur die korrekte Anzahl und die korrekten Typen von Argumenten an eine Prozedur übergeben werden, und erlaubt es Ihrem Programm, Prozeduren aufzurufen, die in separaten Modulen definiert sind. In Abschnitt 2.5.4 finden Sie weitere Informationen zu **DECLARE**.)

Weitere Informationen zur Erstellung von Prozeduren und Modulen mit QuickBASIC finden Sie in Kapitel 6, "Wie Sie QuickBASIC-Programme erstellen und starten" und Kapitel 4, "Wie Sie Quelldateien verwalten", beide in *Lernen und Anwenden von Microsoft QuickBASIC*.

## 2.1 Prozeduren: Bausteine für die Programmierung

In diesem Kapitel umfaßt der Begriff "Prozedur" sowohl die **SUB...END SUB**- als auch die **FUNCTION...END FUNCTION**-Konstruktion. Prozeduren eignen sich zur Zusammenfassung sich wiederholender Aufgaben. Nehmen Sie beispielsweise an, daß Sie ein Programm schreiben, das Sie schließlich als selbständige Anwendung kompilieren wollen, und nehmen Sie weiter an, daß der Benutzer dieser Anwendung die Möglichkeit haben soll, verschiedene Argumente an diese Anwendung von der Befehlszeile aus zu übergeben. Dann ist es sinnvoll, diese Aufgabe – Aufteilung der von der Funktion **COMMAND$** angegebenen Zeichenkette in zwei oder mehr Argumente – in eine Prozedur umzuwandeln. Nachdem Sie diese Prozedur zur Aufteilung der Befehlszeile einmal erstellt und lauffähig gemacht haben, können Sie sie in anderen Programmen verwenden. Im Grunde erweitern Sie BASIC mit der Verwendung von Prozeduren so, daß es Ihre persönlichen Bedürfnisse erfüllt.

Die folgenden zwei Punkte stellen die hauptsächlichen Vorteile der Programmierung mit Prozeduren dar:

1. Prozeduren ermöglichen Ihnen das Aufteilen Ihrer Programme in einzelne logische Einheiten, von denen jede einfacher getestet werden kann, als ein gesamtes Programm ohne Prozeduren.
2. In einem Programm verwendete Prozeduren können in anderen Programmen als Bausteine eingesetzt werden, normalerweise ohne bzw. mit nur kleinen Änderungen.

   Ebenso können Sie Prozeduren in Ihrer eigenen Quick-Bibliothek ablegen. Dies ist eine besondere Datei, die Sie beim Start von QuickBASIC in den Speicher laden können. Wenn sich erst einmal der Inhalt einer Quick-Bibliothek mit QuickBASIC im Speicher befindet, hat jedes Programm, das Sie schreiben, Zugriff auf die Prozeduren in der Bibliothek. Dies vereinfacht es für jedes Ihrer Programme, Programmcode sowohl gemeinsam zu nutzen als auch zu sparen. (In Kapitel 8, "Quick-Bibliotheken", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie weitere Informationen zum Erstellen von Quick-Bibliotheken.)

## 2.2 Prozeduren im Vergleich mit Unterroutinen und Funktionen

Wenn Sie frühere Versionen von BASIC kennen, können Sie sich eine **SUB...END SUB**-Prozedur ähnlich einer **GOSUB...RETURN**-Unterroutine vorstellen. Ebenso werden Sie einige Ähnlichkeiten zwischen einer **FUNCTION...END FUNCTION**-Prozedur und einer **DEF FN...END DEF**-Funktion feststellen. Prozeduren haben jedoch gegenüber diesen älteren Konstruktionen, wie in den Kapiteln 2.2.1 und 2.2.2 gezeigt, einige Vorteile.

***Hinweis*** Um Verwechslungen zwischen **SUB**-Prozeduren und dem Ziel einer **GOSUB**-Anweisung zu vermeiden, welches das *Zeilenkennzeichen*...**RETURN** ist, wird in diesem Handbuch auf **SUB**-Prozeduren mit "Unterprogramm" Bezug genommen, während *Zeilenkennzeichen*...**RETURN**-Blöcke als "Unterroutinen" bezeichnet werden.

## 2.2.1 SUB und GOSUB im Vergleich

Obwohl die **GOSUB**-Unterroutine ein Schritt in die Richtung ist, ein Programm in überschaubare Einheiten aufzuteilen, haben **SUB**-Prozeduren eine Reihe von Vorteilen gegenüber Unterroutinen:

- **Lokale und globale Variablen**

  In **SUB**-Prozeduren sind alle Variablen standardmäßig lokale Variablen; das heißt, sie existieren nur im Geltungsbereich der Definition der **SUB**-Prozedur. Zur Verdeutlichung wird im folgenden Beispiel gezeigt, daß die Variable `I` in dem Unterprogramm `Test` lokal zu `Test` ist, und nichts mit der Variablen `I` im Modul-Ebenen-Code zu tun hat:

```
I = 1
CALL Test
PRINT I   'I ist immer noch gleich 1.
END
SUB Test STATIC
   I = 50
END SUB
```

  Eine **GOSUB**-Unterroutine hat als Baustein im Programm einen erheblichen Nachteil: sie enthält "globale Variablen". Wenn Sie eine Variable `I` innerhalb Ihrer Unterroutine und eine andere Variable namens `I` außerhalb der Unterroutine, aber im gleichen Modul, haben, sind sie ein und dasselbe. Alle Änderungen des Wertes von `I` in der Unterroutine betreffen `I` überall, wo es im Modul erscheint. Das Ergebnis ist, daß Sie beim Versuch des Einbindens einer Unterroutine aus einem anderen Modul zur Vermeidung von Konflikten mit Variablennamen des neuen Modules, die Variablen der Unterroutine eventuell umbenennen müssen.

- **Verwenden in Programmen mit mehreren Modulen**

  Eine **SUB** kann in einem Modul definiert sein und von einem anderen aus aufgerufen werden. Dies hat eine deutliche Einsparung des von einem Programm erforderten Code sowie eine deutliche Vereinfachung in der Verwendung des Codes durch verschiedene Programme zur Folge.

  Eine **GOSUB**-Unterroutine muß im selben Modul definiert und verwendet werden.

- **Arbeiten mit unterschiedlichen Sätzen von Variablen**

  Eine **SUB**-Prozedur kann beliebig oft innerhalb eines Programmes mit einem jeweils anderen Satz von Variablen aufgerufen werden. Dies geschieht durch den Aufruf der **SUB** mit einer Argumentenliste (weitere Informationen hierzu finden Sie in Abschnitt 2.5, "Wie Sie Argumente an Prozeduren übergeben"). Im nächsten Beispiel wird das Unterprogramm `Vergleich` zweimal mit jeweils unterschiedlichen Paaren von Variablen aufgerufen:

```
X = 4 : Y = 5
CALL Vergleich (X,Y)
Z = 7 : W = 2
CALL Vergleich (Z,W)
END
SUB Vergleich (A,B)
   IF A < B THEN SWAP A, B
END SUB
```

  Der Aufruf einer **GOSUB**-Unterroutine mehr als einmal im selben Programm ist schwierig, wenn diese jedesmal einen unterschiedlichen Satz von Variablen bearbeiten soll. Dieser Prozeß bringt das Kopieren von Werten in und aus globalen Variablen mit sich, wie im folgenden Beispiel gezeigt:

```
X = 4 : Y = 5
A = X : B = Y
GOSUB Vergleich
X = A : Y = B
Z = 7 : W = 2
A = Z : B = W
GOSUB Vergleich
Z = A : W = B
END
Vergleich:
   IF A < B THEN SWAP A, B
RETURN
```

## 2.2.2 FUNCTION im Vergleich mit DEF FN

Während die Definition einer vielzeiligen **DEF FN**-Funktion die Antwort auf die Forderung nach komplexeren Aufgaben, als auf einer Zeile untergebracht werden können, ist, bieten Ihnen **FUNCTION**-Prozeduren dieselbe Möglichkeit zusammen mit den unten aufgeführten zusätzlichen Vorteilen:

- **Lokale und globale Variablen**

  Standardmäßig sind innerhalb einer **FUNCTION**-Prozedur alle verwendeten Variablen zu ihr lokal, obwohl Sie die Möglichkeit haben, globale Variablen zu verwenden. (In Abschnitt 2.6, "Die gemeinsame Nutzung von Variablen mit SHARED" finden Sie weitere Informationen zu Prozeduren und globalen Variablen.)

  In einer **DEF FN**-Funktion sind Variablen innerhalb des Körpers der Funktion standardmäßig global zu dem aktuellen Modul (dies gilt ebenso für **GOSUB**-Unterroutinen). In einer **DEF FN**-Funktion können Sie jedoch eine Variable lokal machen, indem Sie sie in eine **STATIC**-Anweisung schreiben.

- **Verändern von Variablen, die an Prozeduren übergeben werden**

  Variablen werden an **FUNCTION**-Prozeduren als Referenz oder als Wert übergeben. Wenn Sie eine Variable als Referenz übergeben, können Sie die Variable durch Änderung ihres zugehörigen Parameters in der **FUNCTION** verändern. Nach dem Aufruf `HoleRest` im nächsten Programm ist zum Beispiel der Wert von `X` zwei, da der Wert von `M` am Ende der **FUNCTION** zwei ist:

```
X = 89
Y = 40
PRINT HoleRest(X,Y)
PRINT X,Y           'X ist jetzt 2
END
FUNCTION HoleRest (M,N)
   HoleRest = M MOD N
   M = M \ N
END FUNCTION
```

Variablen werden an eine **DEF FN**-Funktion nur als Wert übergeben, so daß im nächsten Beispiel `FNRest M` verändert, ohne daß dabei `X` betroffen ist:

```
DEF FNRest (M,N)
   FNRest = M MOD N
   M = M \ N
END DEF
X = 89
Y = 40
PRINT FNRest(X,Y)
PRINT X,Y      'X ist immer noch 89
```

(Weitere Informationen zum Unterschied zwischen Referenzübergabe und Wertübergabe finden Sie in den Abschnitten 2.5.5 und 2.5.6)

- **Aufruf der Prozedur aus ihrer eigenen Definition heraus**

  Eine **FUNCTION**-Prozedur kann "rekursiv" sein, mit anderen Worten, sie kann sich aus ihrer eigenen Definition heraus aufrufen. (Weitere Informationen zu rekursiven Prozeduren finden Sie in Abschnitt 2.9.)

  Eine **DEF FN**-Funktion kann nicht rekursiv sein.

- **Verwendung in Programmen mit mehreren Modulen**

  Sie können eine **FUNCTION**-Prozedur in einem Modul definieren und in einem anderen Modul verwenden. Sie müssen in dem Modul, in dem die **FUNCTION** verwendet wird, eine **DECLARE**-Anweisung einsetzen; andernfalls nimmt Ihr Programm an, daß sich der **FUNCTION**-Name auf eine Variable bezieht. (In Abschnitt 2.5.4, "Überprüfen von Argumenten mit der Anweisung **DECLARE**" finden Sie weitere Informationen zu der Benutzung der **DECLARE**-Anweisung für diesen Zweck.)

  Sie müssen eine **DEF FN**-Funktion in demselben Modul definieren und verwenden. Zusätzlich muß die **DEF FN**-Funktion *immer* definiert sein, bevor sie in einem Modul verwendet wird.

- **Einschränkungen für die Benennung**

  Der Name einer **FUNCTION**-Prozedur kann jeder gültige BASIC-Variablenname sein, der nicht mit den Buchstaben "FN" beginnt.

  Dem Namen einer **DEF FN**-Funktion muß immer "FN" voranstehen.

## 2.3 Wie Sie Prozeduren definieren

BASIC-Prozedurdefinitionen haben die folgende allgemeine Syntax:

{**SUB** | **FUNCTION**} *Prozedurname* [(*Parameterliste*)] [**STATIC**]
.
.
Prozedurdefinition:
ein Block von Anweisungen, der angibt, was die Prozedur tut
.
.
[**EXIT** {**SUB** | **FUNCTION**}]
.
.
**END** {**SUB** | **FUNCTION**}

Die folgende Liste beschreibt die Teile einer Prozedurdefinition:

| ***Teil*** | ***Beschreibung*** |
|---|---|
| {**SUB** \| **FUNCTION**} | Markiert den Anfang einer **SUB**- oder **FUNCTION**-Prozedur. |
| *Prozedurname* | Jeder gültige Variablenname mit bis zu 40 Zeichen. Derselbe *Prozedurname* kann nicht sowohl für eine **SUB** als auch **FUNCTION** verwendet werden. |
| *Parameterliste* | Eine Liste von Variablen, die durch Kommata getrennt sind. Die *Parameterliste* enthält die Anzahl und Typen der Argumente, die an die Prozedur übergeben werden (in Abschnitt 2.5.1 finden Sie eine Erläuterung zu dem Unterschied von Parametern und Argumenten). |

| *Teil* | *Beschreibung* |
|---|---|
| **STATIC** | Legt fest, wie lokale Variablen einer Prozedur standardmäßig im Speicher abgelegt werden.<br><br>Wenn Sie das Attribut **STATIC** verwenden, sind lokale Variablen **STATIC**; das heißt, sie behalten zwischen Prozeduraufrufen ihren Wert.<br><br>Wenn Sie das Attribut **STATIC** auslassen sind lokale Variablen standardmäßig "automatisch", das heißt, sie werden zum Beginn jedes Prozeduraufrufs zu Null oder zu Null-Zeichenketten initialisiert.<br><br>In Abschnitt 2.7, "Automatische und STATIC-Variablen", in diesem Handbuch und in Kapitel 2, "Datentypen", im *BASIC-Befehlsverzeichnis* finden Sie weitere Informationen. |
| **END {SUB \| FUNCTION}** | Beendet die Definition.<br><br>**END SUB** markiert das Ende einer **SUB**-Definition, während **END FUNCTION** das Ende einer **FUNCTION**-Definition markiert. Um korrekt zu laufen, muß jede Prozedur genau eine **END {SUB \| FUNCTION}**-Anweisung haben.<br><br>Wenn Ihr Programm ein **END SUB** oder **END FUNCTION** erreicht, verläßt es die Prozedur und springt zu der Anweisung zurück, die direkt auf diejenige folgt, die die Prozedur aufgerufen hat. Ebenso können Sie eine oder mehrere optionale **EXIT {SUB \| FUNCTION}**-Anweisungen innerhalb des Körpers der Prozedurdefinition verwenden, um die Prozedur zu verlassen. |

Innerhalb einer Prozedurdefinition sind alle gültigen BASIC-Ausdrücke und -Anweisungen erlaubt, außer den folgenden:

- **DEF FN...END DEF, FUNCTION...END FUNCTION** oder **SUB...END SUB.**

  Mit anderen Worten, es ist nicht möglich, Prozedurdefinitionen zu verschachteln oder eine **DEF FN**-Funktion innerhalb einer Prozedur zu definieren. Eine Prozedur kann jedoch eine andere Prozedur oder **DEF FN**-Funktion aufrufen.
- **COMMON**
- **DECLARE**
- **DIM SHARED**
- **OPTION BASE**
- **TYPE...END TYPE**

**Beispiel**

Das folgende Beispiel zeigt die Definition für eine **FUNCTION**-Prozedur mit dem Namen `GanzZahlPot`:

```
FUNCTION GanzZahlPot& (X&, Y&) STATIC
   PotenzWert& = 1
   FOR I& = 1 TO Y&
      PotenzWert& = PotenzWert& * X&
   NEXT I&
   GanzZahlPot& = PotenzWert&
END FUNCTION
```

## 2.4 Wie Sie Prozeduren aufrufen

In den nächsten beiden Abschnitten wird gezeigt, daß sich der Aufruf einer **FUNCTION**-Prozedur vom Aufruf einer **SUB**-Prozedur unterscheidet.

### 2.4.1 Wie Sie eine FUNCTION-Prozedur aufrufen

Eine **FUNCTION**-Prozedur rufen Sie auf die gleiche Weise auf wie eine eingebaute BASIC-Funktion (z. B. **ABS**): Das heißt, Sie verwenden ihren Namen, wie hier gezeigt, in einem Ausdruck:

```
' Jede der folgenden Anweisungen würde eine FUNCTION
' des Namens "BisZehn" aufrufen :
PRINT 10 * BisZehn
X = BisZehn
IF BisZehn = 10 THEN PRINT "Außerhalb des Bereichs."
```

Eine **FUNCTION** kann Werte zurückgeben, indem sie die ihr als Argumente übergebenen Variablen ändert (eine Erläuterung, warum dies passiert, finden Sie in Abschnitt 2.5.5, "Wie Sie Argumente als Referenz übergeben"). Zusätzlich gibt eine **FUNCTION** in ihrem Namen einen einzigen Wert an, so daß der Name einer **FUNCTION** mit dem Typ, den sie zurückgibt, übereinstimmen muß. Wenn zum Beispiel eine **FUNCTION** einen Zeichenkettenwert angibt, muß ihr Name entweder ein angehängtes Dollar-Zeichen ($) als Typdeklarationszeichen haben, oder sie muß in einer vorhergehenden **DEFSTR**-Anweisung als zum Typ Zeichenkette gehörig deklariert sein.

### Beispiel

Das folgende Programm zeigt eine **FUNCTION**, die einen Zeichenkettenwert angibt. Beachten Sie, daß das Typdeklarationssuffix für Zeichenketten ($) Teil des Prozedurnamens ist.

```
Eingabe$ = HolEingabe$ 'Ruf die FUNCTION auf und weise
                       'den zurückgegebenen Wert der
                       'Zeichenkettenvariablen zu.
PRINT Eingabe$         'Gib die Zeichenkette aus.
END
'
' ================== HOLEINGABE$ ====================
'    Das Typdeklarationszeichen $ am Ende des Namens
'    dieser FUNCTION bedeutet, daß sie einen
'    Zeichenkettenwert angibt.
' ====================================================
'
FUNCTION HolEingabe$ STATIC
   ' Gib eine Zeichenkette mit 10 von der Tastatur
   ' gelesenen Zeichen an. Jedes Zeichen wird
   ' geschrieben, sobald es eingegeben ist:
   FOR I% = 1 TO 10
      Zeich$ = INPUT$(1) 'Hole das Zeichen.
      PRINT Zeich$;      'Schreib das Zeichen auf den
                         'Bildschirm.
      Temp$ = Temp$ + Zeich$ 'Füge das Zeichen der
                             'Zeichenkette hinzu.
   NEXT
   PRINT
   HolEingabe$ = Temp$   'Weise die Zeichenkette der
                         'FUNCTION zu.
END FUNCTION
```

## 2.4.2 Wie Sie eine SUB-Prozedur aufrufen

Eine **SUB**-Prozedur unterscheidet sich von einer **FUNCTION**-Prozedur dadurch, daß eine **SUB** nicht aufgerufen werden kann, indem ihr Name innerhalb eines Ausdruckes verwendet wird. Der Aufruf einer **SUB** ist eine Anweisung für sich, wie z. B. die BASIC-Anweisung **CIRCLE**. Ebenso gibt eine **SUB** einen Wert nicht mit ihrem Namen zurück, wie dies eine **FUNCTION** tut. Eine **SUB** kann jedoch wie eine **FUNCTION** die Werte der an sie übergebenen Variablen verändern (eine Erläuterung, warum dies passiert, finden Sie in Abschnitt 2.5.5, "Wie Sie Argumente als Referenz übergeben").

Sie können eine **SUB** auf eine der folgenden zwei Arten aufrufen:

1. Schreiben Sie ihren Namen in eine **CALL**-Anweisung:

```
CALL GibNachricht
```

2. Benutzen Sie ihren Namen als eigene Anweisung:

```
GibNachricht
```

Wenn Sie sich dazu entscheiden, das Schlüsselwort **CALL** auszulassen, dürfen Sie die an die **SUB** übergebenen Argumente nicht in Klammern einschließen:

```
' Rufe das Unterprogramm BearbEing mit CALL auf und
' übergib die drei Argumente Erstens$, Zweitens$ und
' AnzArg% an das Unterprogramm:
CALL BearbEing (Erstens$, Zweitens$, AnzArg%)
' Rufe das Unterprogramm BearbEing ohne CALL auf und
' übergib ihr die gleichen Argumente (Beachte: die
' Argumentenliste nicht in Klammern einschließen):
BearbEing Erstens$, Zweitens$, AnzArg%
```

Weitere Informationen zur Übergabe von Argumenten an Prozeduren finden Sie in Abschnitt 2.5.

Wenn Ihr Programm **SUB**-Prozeduren aufruft, ohne **CALL** zu verwenden, und wenn Sie nicht QuickBASIC einsetzen, um Ihr Programm zu schreiben, dann müssen Sie den Namen der **SUB** in eine **DECLARE**-Anweisung schreiben, bevor die **SUB** aufgerufen wird.

```
DECLARE SUB TestAufTastatur
.
.
.
TestAufTastatur
```

Darauf müssen Sie nur achten, wenn Sie Programme außerhalb von QuickBASIC entwickeln, weil QuickBASIC beim Speichern von Programmen überall dort, wo sie benötigt werden, automatisch **DECLARE**-Anweisungen einfügt.

# 2.5 Wie Sie Argumente an Prozeduren übergeben

Die Abschnitte 2.5.1 bis 2.5.4 erläutern, wie Parameter und Argumente unterschieden werden, wie Argumente an Prozeduren übergeben werden und Argumente überprüft werden, um sicherzustellen, daß sie den korrekten Typ und die korrekte Anzahl haben.

## 2.5.1 Parameter und Argumente

Der erste Schritt, die Übergabe von Argumenten an Prozeduren zu lernen, ist das Verstehen des Unterschiedes zwischen den Begriffen "Parameter" und "Argument".

| ***Parameter*** | ***Argument*** |
|---|---|
| Ein Variablenname, der in einer **SUB**-, **FUNCTION**- oder **DECLARE**-Anweisung erscheint | Eine Konstante, Variable oder ein Ausdruck, die/der an eine **SUB** oder **FUNCTION** übergeben wird, wenn die **SUB** oder **FUNCTION** aufgerufen wird |

In einer Prozedurdefinition sind Parameter Platzhalter für Argumente. Wie Abbildung 2.1 zeigt, werden beim Aufruf einer Prozedur Argumente in die Variablen der Parameterliste übergeben, wobei der erste Parameter das erste Argument erhält, der zweite Parameter das zweite Argument erhält usw.

*Abbildung 2.1 Parameter und Argumente*

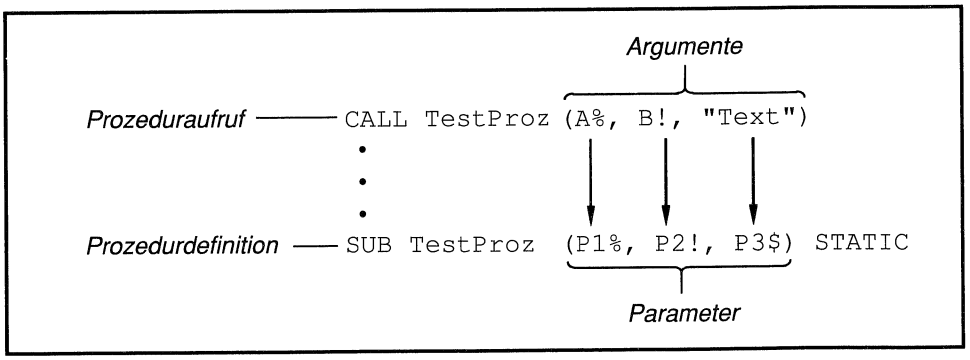


Abbildung 2.1 demonstriert ebenso eine weitere wichtige Regel: Obwohl die Namen von Variablen in einer Argumentenliste und einer Parameterliste nicht gleich sein müssen, muß die Anzahl von Parametern und Argumenten übereinstimmen. Darüberhinaus sollte der Typ (Zeichenkette, Ganzzahl, Zahl einfacher Genauigkeit usw.) für zusammengehörende Argumente und Parameter gleich sein. (Weitere Informationen darüber, wie sichergestellt wird, daß Argumente und Parameter in Anzahl und Typ übereinstimmen, finden Sie in 2.5.4, "Überprüfen von Argumenten mit der Anweisung **DECLARE**".)

Eine Parameterliste setzt sich zusammen aus (jeweils durch Kommata getrennt):

- Gültigen Variablennamen, mit Ausnahme von Zeichenketten fester Länge.

  Mit anderen Worten, `x$` und `x AS STRING` sind beide gültig in einer Parameterliste, da sie sich auf Zeichenketten variabler Länge beziehen. Jedoch verweist `x AS STRING * 10` auf eine 10 Zeichen lange Zeichenkette fester Länge und kann nicht in einer Parameterliste erscheinen. (Zeichenkette fester Länge können aber als *Argumente* an Prozeduren übergeben werden. Weitere Informationen über Zeichenkette fester- und variabler Länge finden Sie in Kapitel 4, "Zeichenkettenverarbeitung".)
- Datenfeldnamen, denen linke und rechte Klammern folgen,

Eine Argumentenliste setzt sich zusammen aus (jeweils durch Kommata getrennt):

- Konstanten
- Ausdrücken
- gültigen Variablennamen
- Datenfeldnamen, denen linke und rechte Klammern folgen,

### Beispiele

Das folgende Beispiel zeigt die erste Zeile einer Unterprogrammdefinition mit einer Parameterliste:

```
SUB TestSub (A%, DatFeld(), VerbVar AS VerbTyp, Cs$)
```

Der erste Parameter, `A%`, ist eine Ganzzahl; der zweite Parameter, `DatFeld()`, ist ein Datenfeld einfacher Genauigkeit, da nichttypisierte Variablen standardmäßig von einfacher Genauigkeit sind; der dritte Parameter, `VerbVar`, ist ein Verbund vom Typ `VerbTyp`; und der vierte Parameter `Cs$` ist eine Zeichenkette.

Die Zeile `CALL TestSub` im nächsten Beispiel ruft das Unterprogramm `TestSub` auf und übergibt dem Unterprogramm vier Argumente des passenden Typs:

```
TYPE VerbTyp
   Reihe     AS STRING * 12
   SerienNr  AS LONG
END TYPE
DIM VerbVar AS VerbTyp
CALL TestSub (X%, A(), VerbVar, "Gummibaum")
```

## 2.5.2 Wie Sie Konstanten und Ausdrücke übergeben

Konstanten – entweder Zeichenketten oder numerische – können in der Liste der an die Prozedur übergebenen Argumente erscheinen. Selbstverständlich muß eine Zeichenkettenkonstante an einen Zeichenkettenparameter übergeben werden, und eine numerische Konstante an einen numerischen Parameter, wie im nächsten Beispiel gezeigt:

```
CONST BILDBREITE = 80
CALL UebSchrift (BILDBREITE, "Bestands-Bericht")
.
.
.
SUB UebSchrift (BD%, Titel$)
    .
    .
    .
END SUB
```

Wenn eine numerische Konstante in einer Argumentenliste nicht den gleichen Typ hat wie der zugehörige Parameter in der **SUB**- oder **FUNCTION**-Anweisung, dann wird die Konstante in den Typ des Parameters umgewandelt, wie Sie in der Ausgabe des nächsten Beispiels sehen können:

```
CALL test(4.6, 4.1)
END
SUB test (X%, Y%)
   PRINT X%, Y%
END SUB
```

***Ausgabe***

```
 5            4
```

Aus Operationen mit Variablen und Konstanten resultierende Ausdrücke können ebenso an eine Prozedur übergeben werden. Genauso wie bei Konstanten werden numerische Ausdrücke, die im Typ mit ihren Parametern nicht übereinstimmen, entsprechend umgewandelt, wie nachfolgend gezeigt:

```
Pruefe A!+25!, NOT BooleWert%
'Im nächsten Aufruf wandelt die Angabe von Klammern um
'die langganzzahlige Variable BWert& diese in einen
'Ausdruck um.
'Der Ausdruck (BWert&) wird in der SUB Pruefe in eine
'kurze Ganzzahl umgewandelt:
Pruefe A!/3.1, (BWert&)
.
.
.
END
SUB Pruefe (Param1!, Param2%)
    .
    .
    .
END SUB
```

## 2.5.3 Wie Sie Variablen übergeben

Dieser Abschnitt beschreibt, wie einfache Variablen, komplette Datenfelder, Elemente von Datenfeldern, Verbunde und Elemente von Verbunden an Prozeduren übergeben werden.

### 2.5.3.1 Wie Sie einfache Variablen übergeben

Sowohl in einer Argumentenliste als auch in einer Parameterliste können Sie den Typ für eine einfache Variable auf eine der drei folgenden Arten deklarieren:

1. Hinzufügen eines der folgenden Typdeklarationssuffixe zum Variablennamen:

   % & ! # $

2. Deklarieren der Variablen in einer Klausel *Deklarieren Variablenname* **AS** *Typ*, wobei der Platzhalter *Deklarieren* entweder **DIM, COMMON, REDIM, SHARED** oder **STATIC** sein kann, und *Typ* entweder **INTEGER, LONG, SINGLE, DOUBLE, STRING** oder **STRING** * *n* sein kann. Zum Beispiel:

   ```
   DIM A AS LONG
   ```

3. Verwendung einer **DEF***Typ*-Anweisung, um den Standardtyp zu setzen (weitere Informationen zur **DEF***Typ*-Anweisung finden Sie im *BASIC-Befehlsverzeichnis*).

Ganz gleich, welche Methode Sie wählen, zusammengehörende Variablen müssen denselben Typ sowohl in den Argumentenlisten als auch in den Parameterlisten haben, wie im folgenden Beispiel gezeigt:

#### Beispiel

In diesem Beispiel werden zwei Argumente an die **FUNCTION**-Prozedur übergeben. Das erste ist eine Ganzzahl, die die Länge der von der **FUNCTION** angegebenen Zeichenkette anzeigt, während das zweite ein Zeichen ist, das zur Bildung der Zeichenkette wiederholt wird.

```
FUNCTION ZeichKett$(A AS INTEGER, B$) STATIC
   ZeichKett$ = STRING$(A%,B$)
END FUNCTION
DIM X AS INTEGER
INPUT "Geben Sie eine Zahl ein (1 bis 80): ", X
INPUT "Geben Sie ein Zeichen ein: ", Y$
' Gib eine Zeichenkette, die aus dem Zeichen in Y$
' besteht, X-mal aus:
PRINT ZeichKett$(X, Y$)
END
```

***Ausgabe***

```
Geben Sie eine Zahl ein (1 bis 80): 21
Geben Sie ein Zeichen ein: #
####################
```

### 2.5.3.2 Wie Sie ein vollständiges Datenfeld übergeben

Um alle Elemente eines Datenfeldes an eine Prozedur zu übergeben, schreiben Sie den Namen des Datenfeldes - gefolgt von linken und rechten Klammern - sowohl in die Argumentenliste als auch in die Parameterliste.

**Beispiel**

Das folgende Beispiel zeigt, wie alle Elemente eines Datenfeldes an eine Prozedur übergeben werden:

```
DIM Werte(1 TO 5) AS INTEGER
' Beachte die leeren Klammern nach den Datenfeldnamen
' bei Prozeduraufruf und Übergabe der Datenfelder:
CALL WechsDatF (1, 5, Werte())
CALL AusgDatF (1, 5, Werte())
END
' Beachte die leeren Klammern nach dem Parameter P:
'
SUB WechsDatF (Min%, Max%, P() AS INTEGER) STATIC
   FOR I% = Min% TO Max%
      P(I%) = I% ^ 3
   NEXT I%
END SUB
SUB AusgDatF (Min%, Max%, P() AS INTEGER) STATIC
   FOR I% = Min% TO Max%
      PRINT P(I%),
   NEXT I%
   PRINT
END SUB
```

### 2.5.3.3 Wie Sie einzelne Datenfeldelemente übergeben

Wenn eine Prozedur kein vollständiges Datenfeld benötigt, können Sie stattdessen einzelne Elemente dieses Datenfeldes übergeben. Um ein Element eines Datenfeldes zu übergeben, verwenden Sie den Datenfeldnamen, gefolgt von dem in Klammern eingeschlossenen passenden Index.

### Beispiel

Die Anweisung `WurzWert DatFeld(5,3)` im folgenden Beispiel übergibt das Element in Zeile 5, Spalte 3 von `DatFeld` an das Unterprogramm `WurzWert` (Beachten Sie, wie das Unterprogramm nun den Wert dieses Datenfeldelementes verändert):

```
DIM DatFeld(1 TO 5,1 TO 3)
DatFeld(5,3) = -36
PRINT DatFeld(5,3)
WurzWert DatFeld(5,3)
PRINT DatFeld(5,3) ' Der Aufruf von WurzWert hat den
                   ' Wert von DatFeld(5,3) verändert.
END
SUB WurzWert(A) STATIC
   A = SQR(ABS(A))
END SUB
```

### *Ausgabe*

```
-36
 6
```

## 2.5.3.4 Wie Sie die Funktionen für Datenfeldgrenzen verwenden

Die Funktionen **LBOUND** und **UBOUND** sind hilfreich für die Bestimmung der Größe eines an eine Prozedur übergebenen Datenfeldes. Die Funktion **LBOUND** ermittelt den kleinsten Indexwert eines Datenfeldindexes, während die Funktion **UBOUND** den größten Indexwert ermittelt. Diese Funktionen bewahren Sie vor dem Problem, die untere und obere Grenze für jede Datenfelddimensionierung an eine Prozedur übergeben zu müssen.

### Beispiel

Das Unterprogramm im folgenden Beispiel verwendet die **LBOUND**-Funktion, um die Variablen `Zeile` und `Spalte` auf die jeweilig kleinsten Indexwerte der Dimensionen von `A` zu initialisieren. Darüberhinaus verwendet es die Funktion **UBOUND**, um die Anzahl der Wiederholungen der `FOR`-Schleife auf die Anzahl der Elemente in dem Datenfeld zu begrenzen.

```
SUB Ausgabe(A(2)) STATIC
   FOR Zeile = LBOUND(A,1) TO UBOUND(A,1)
      FOR Spalte = LBOUND(A,2) TO UBOUND(A,2)
          PRINT A(Zeile,Spalte)
      NEXT Spalte
   NEXT Zeile
END SUB
```

### 2.5.3.5 Wie Sie einen vollständigen Verbund übergeben

Um einen kompletten Verbund (eine Variable, die als benutzerdefinierter Typ deklariert ist) an eine Prozedur zu übergeben, gehen Sie wie folgt vor:

1. Definieren Sie den Typ (in diesem Beispiel `WarenBest`):

```
TYPE WarenBest
   TeilNum     AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE
```

2. Deklarieren Sie eine Variable (`WarenSatz`) mit diesem Typ:

```
DIM WarenSatz AS WarenBest
```

3. Rufen Sie eine Prozedur (`FindeSatz`) auf und übergeben dieser die von Ihnen deklarierte Variable:

```
CALL FindeSatz (WarenSatz)
```

4. Geben Sie in der Prozedurdefinition dem Parameter den gleichen Typ wie der Variablen:

```
SUB FindeSatz (SatzVar AS WarenBest) STATIC
   .
   .
   .
END SUB
```

#### 2.5.3.6 Wie Sie einzelne Elemente eines Verbundes übergeben

Um einzelne Elemente eines Verbundes an eine Prozedur zu übergeben, schreiben Sie den Namen des Elementes (*Verbundname.Elementname*) in die Argumentenliste. Vergewissern Sie sich, wie immer, daß der zugehörige Parameter in der Prozedurdefinition mit dem Typ dieses Elementes übereinstimmt.

#### Beispiel

Das folgende Beispiel zeigt, wie zwei Elemente der Verbundvariablen `WarenSatz` an die **SUB**-Prozedur `SchrPreisEtik` übergeben werden. Beachten Sie, wie jeder Parameter in der **SUB**-Prozedur mit dem Typ der einzelnen Verbundelemente übereinstimmt.

```
TYPE WarenBest
   TeilNum     AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE
DIM WarenSatz AS WarenBest
CALL SchrPreisEtik (WarenSatz.Beschreib,WarenSatz.EinzPreis)
.
.
.
END
SUB SchrPreisEtik (Beschr$, Preis AS SINGLE)
    .
    .
    .
END SUB
```

### 2.5.4 Überprüfen von Argumenten mit der Anweisung DECLARE

Wenn Sie QuickBASIC zum Schreiben Ihrer Programme verwenden, werden Sie bemerken, daß QuickBASIC beim Speichern des Programmes für jede Prozedur automatisch eine **DECLARE**-Anweisung einfügt. Jede **DECLARE**-Anweisung besteht aus dem Wort **DECLARE**, gefolgt von den Worten **SUB** oder **FUNCTION**, dem Namen der Prozedur und einem Paar von Klammern. Die Klammern sind leer, wenn die Prozedur keine Parameter hat. Hat die Prozedur Parameter, dann umschließen die Klammern eine *Parameterliste*, die die Anzahl und Typen der an die Prozedur zu übergebenden Argumente angibt. Diese *Parameterliste* hat dasselbe Format wie die *Parameterliste* in der Definitionszeile der **SUB** oder **FUNCTION**.

Der Sinn der *Parameterliste* in einer **DECLARE**-Anweisung ist es, eine "Typüberprüfung" der an die Prozedur übergebenen Parameter einzuschalten. Das heißt, jedesmal wenn die Prozedur mit Variablenargumenten aufgerufen wird, werden diese Variablen überprüft um sicher zu gehen, daß sie mit der Anzahl und den Typen der Parameter in der **DECLARE**-Anweisung übereinstimmen.

QuickBASIC schreibt, wenn es ein Programm speichert, alle Prozedurdefinitionen an das Ende eines Modules. Daher würden Sie ohne das Vorhandensein von **DECLARE**-Anweisungen bei dem Versuch, dieses Programm mit dem Befehl **bc** zu kompilieren, auf das als "Vorwärtsbezug" (eine Prozedur wird aufgerufen, bevor sie definiert ist) bekannte Problem stoßen. Indem sie ein Muster der Prozedurdefinition erstellt, erlaubt die Anweisung **DECLARE** Ihrem Programm den Aufruf von Prozeduren, die später in dem Modul oder in einem völlig anderen Modul definiert sind.

### Beispiel

Das nächste Beispiel zeigt in der **DECLARE**-Anweisung eine leere Parameterliste, da an `HolEingabe$` keine Argumente übergeben werden:

```
DECLARE FUNCTION HolEingabe$ ()
X$ = HolEingabe$
FUNCTION HolEingabe$ STATIC
   HolEingabe$ = INPUT$(10)
END FUNCTION
```

Das nächste Beispiel zeigt in der **DECLARE**-Anweisung eine Parameterliste, da an diese Version von `HolEingabe$` ein Ganzzahlargument übergeben wird:

```
DECLARE FUNCTION HolEingabe$ (X%)
X$ = HolEingabe$ (5)
FUNCTION HolEingabe$ (X%) STATIC
   HolEingabe$ = INPUT$(X%)
END FUNCTION
```

### 2.5.4.1 Wann QuickBASIC keine DECLARE-Anweisung erzeugt

In einigen Fällen erzeugt QuickBASIC keine **DECLARE**-Anweisungen in dem Modul, das eine Prozedur aufruft.

QuickBASIC kann in einem Modul keine **DECLARE**-Anweisung für eine **FUNCTION**-Prozedur erzeugen, die in einem anderen Modul definiert ist. Wenn Ihr Programm ein Modul enthält, das eine **FUNCTION**-Prozedur eines anderen Modules aufruft, müssen Sie die **DECLARE**-Anweisung selbst an den Anfang des Modules, in dem die **FUNCTION** aufgerufen wird, schreiben; anderenfalls betrachtet QuickBASIC den Aufruf der **FUNCTION** lediglich als einen Variablennamen.

Für eine Prozedur in einer Quick-Bibliothek kann QuickBASIC ebenfalls keine **DECLARE**-Anweisung erzeugen.

### 2.5.4.2 Wie Sie Programme außerhalb der QuickBASIC-Umgebung entwickeln

Wenn Sie Ihre Programme mit Ihrem eigenen Texteditor schreiben und anschließend mit den Befehlen **bc** und **link** außerhalb der QuickBASIC-Umgebung kompilieren, müssen Sie an den folgenden drei Stellen **DECLARE**-Anweisungen schreiben:

1. Am Beginn eines jeden Moduls, das eine **FUNCTION**-Prozedur aufruft, bevor diese definiert ist:

```
DECLARE FUNCTION Hypot (X!, Y!)
INPUT X, Y
PRINT Hypot(X, Y)
END
FUNCTION Hypot (A, B) STATIC
   Hypot = SQR(A ^ 2 + B ^ 2)
END FUNCTION
```

2. Am Beginn eines jeden Moduls, das eine **SUB**-Prozedur aufruft, bevor diese definiert ist, und beim Aufruf der **SUB** kein **CALL** verwendet:

```
DECLARE SUB AusgabeZeichK (X,Y)
INPUT X,Y
AusgabeZeichK X, Y      ' Beachte: keine Klammern um
                        ' die Argumente
END
SUB AusgabeZeichK(A,B) STATIC
   ' Übertrage die Zahlen in Zeichenketten, entferne
   ' alle führenden Leerzeichen aus der zweiten Zahl
   ' und gib aus:
   PRINT STR$(A) + LTRIM$(STR$(B))
END SUB
```

Wenn Sie eine **SUB**-Prozedur mit **CALL** aufrufen, müssen Sie die **SUB** nicht zuerst deklarieren:

```
A$ = "466"
B$ = "123"
CALL AusgabeZeichK(A$, B$)
END

SUB AusgabeZeichK(X$, Y$) STATIC
   PRINT VAL(X$) + VAL(Y$)
END SUB
```

3. Am Beginn eines jeden Moduls, das eine **SUB**- oder **FUNCTION**-Prozedur aufruft, die in einem anderen Modul definiert ist (eine "externe Prozedur").

Wenn Ihre Prozedur keine Parameter hat, vergessen Sie nicht, in der **DECLARE**-Anweisung nach dem Namen der Prozedur leere Klammern zu schreiben, wie in dem nächsten Beispiel gezeigt:

```
DECLARE FUNCTION LiesStunde$ ()
PRINT LiesStunde$
END

FUNCTION LiesStunde$ STATIC
   LiesStunde$ = LEFT$(TIME$,2)
END FUNCTION
```

Erinnern Sie sich daran, daß eine **DECLARE**-Anweisung nur auf der Modul-Ebene erscheinen kann, nicht auf der Prozedur-Ebene. Eine **DECLARE**-Anweisung hat Auswirkung auf das gesamte Modul, in dem sie erscheint.

Eine eingehendere Erläuterung der Syntax und aller der mit **DECLARE** verknüpften Optionen finden Sie in der Beschreibung der **DECLARE**-Anweisung im *BASIC-Befehlsverzeichnis.*

### 2.5.4.3 Wie Sie Include-Dateien für Deklarationen verwenden

Wenn Sie ein separates Definitionsmodul, in dem eine oder mehrere **SUB**- oder **FUNCTION**-Prozeduren definiert werden, angelegt haben, ist es vorteilhaft, eine dazu passende Include-Datei anzulegen. Diese Include-Datei sollte folgendes enthalten:

- **DECLARE**-Anweisungen für alle Prozeduren des Moduls.
- **TYPE...END TYPE**-Verbunddefinitionen für alle Verbundparameter der **SUB**- oder **FUNCTION**-Prozeduren in diesem Modul.
- **COMMON**-Anweisungen, die die von diesem Modul und anderen Modulen in dem Programm gemeinsam benutzten Variablen auflisten (weitere Informationen über die Verwendung von **COMMON** für diese Aufgabe finden Sie in Abschnitt 2.6.3, "Die gemeinsame Nutzung von Variablen mit anderen Modulen").

Jedesmal, wenn Sie das Definitionsmodul in einem Ihrer Programme verwenden, fügen Sie zu Beginn jedes Moduls, das Prozeduren des Definitionsmoduls aufruft, einen **$INCLUDE**-Metabefehl ein. Wenn Ihr Programm kompiliert ist, ist der **$INCLUDE**-Metabefehl durch die aktuellen Inhalte der Include-Datei ersetzt.

Eine einfache Faustregel ist, daß für jedes Modul eine Include-Datei erstellt wird, und anschließend das Modul und die Include-Datei, wie zuvor beschrieben, zusammen verwendet werden. Die folgende Liste führt einige der Vorteile dieser Technik einzeln auf:

- Ein Modul, das Prozedurdefinitionen enthält, bleibt wirklich modular - das heißt, Sie müssen nicht jedesmal alle **DECLARE**-Anweisungen für die Prozeduren des Modules kopieren, wenn Sie diese aus einem anderen Modul heraus aufrufen; stattdessen genügt es, einen **$INCLUDE**-Metabefehl einzusetzen.
- Wenn Sie QuickBASIC verwenden, unterdrückt die Benutzung einer Include-Datei für Prozedurdeklarationen die automatische Erzeugung von **DECLARE**-Anweisungen beim Speichern des Programms.
- Die Verwendung einer Include-Datei für Deklarationen vermeidet Probleme bei der Erkennung einer **FUNCTION** in einem Modul durch ein anderes Modul. (Weitere Informationen finden Sie in Abschnitt 2.5.4.1 "Wann QuickBASIC keine **DECLARE**-Anweisung erzeugt".)

Sie können sogar die Einrichtung von QuickBASIC, **DECLARE**-Anweisungen zu erzeugen, nutzen, wenn Sie Ihre Include-Datei anlegen. Die folgenden Schritte zeigen Ihnen, wie dies vor sich geht:

1. Erstellen Sie Ihr Modul.
2. Rufen Sie innerhalb dieses Moduls irgendeine von Ihnen definierte **SUB**- oder **FUNCTION**-Prozedur auf.
3. Speichern Sie das Modul, um für alle der Prozeduren automatisch **DECLARE**-Anweisungen zu erhalten.
4. Editieren Sie das Modul erneut, indem Sie alle Prozeduraufrufe entfernen und die **DECLARE**-Anweisungen in eine separate Include-Datei verschieben.

In Kapitel 4, "Wie Sie Quelldateien verwalten", in *Lernen und Anwenden von Microsoft QuickBASIC*, können Sie lernen, wie Include-Dateien in der QuickBASIC-Umgebung erstellt, geladen und editiert werden. Weitere Informationen zur Syntax und Verwendung des Metabefehls **$INCLUDE** finden Sie im Anhang C, "Metabefehle", im *BASIC-Befehlsverzeichnis*.

### Beispiel

Der folgende Ausschnitt illustriert, wie ein Definitionsmodul und eine Include-Datei zusammen verwendet werden:

```
' ====================================================
'                   MODDEF.BAS
'   Dieses Modul enthält Definitionen für die
'           Prozeduren ANFRAGE und MAX!.
' ====================================================
'
FUNCTION Max! (X!, Y!) STATIC
   IF X! > Y! THEN Max! = X! ELSE Max! = Y!
END FUNCTION

SUB Anfrage (Zeile%, Spalte%, SatzVar AS SatzTyp)  STATIC
   LOCATE Zeile%, Spalte%
   INPUT "Beschreibung: ", SatzVar.Beschreibung
   INPUT "Menge:        ", SatzVar.Menge
END SUB
```

```
'
' ====================================================
'                    MODDEF.BI
' Dies ist eine Include-Datei, die DECLARE-Anweisungen
' enthält für die Prozeduren ANFRAGE und MAX! (genauso
' wie eine TYPE-Anweisung, die den Benutzertyp SatzTyp
' definiert). Verwenden Sie diese Datei immer dann,
' wenn Sie das Modul MODDEF.BAS einsetzen.
' ====================================================
'
TYPE SatzTyp
   Beschreibung AS STRING * 15
   Menge        AS INTEGER
END TYPE
DECLARE FUNCTION Max! (X!, Y!)
DECLARE SUB Anfrage (Zeile%, Spalte%, SatzVar AS SatzTyp)
'
' ====================================================
'                BEISP.BAS
' Dieses Modul ist mit dem Modul MODDEF.BAS gebunden
' und ruft die Prozeduren ANFRAGE und MAX! in
' MODDEF.BAS auf.
' ====================================================
'
' Die nächste Zeile läßt die Inhalte der Include-Datei
' MODDEF.BI ebenfalls Teil dieses Moduls werden:
' $INCLUDE: 'MODDEF.BI'
.
.
.
INPUT A, B
PRINT Max!(A, B)       'Rufe die FUNCTION Max! in
                       'MODDEF.BAS auf.
.
.
.
Anfrage 5, 5, SatzVar 'Rufe die SUB Anfrage in
                       'MODDEF.BAS auf.
.
.
.
```

***Wichtig*** Während es eine gute Programmierpraxis ist, Prozedur-*Deklarationen* in eine Include-Datei zu schreiben, dürfen Sie keine Prozedur-*Definitionen* (**SUB...END SUB** oder **FUNCTION...END FUNCTION**-Blöcke) in eine Include-Datei schreiben. Prozedurdefinitionen sind in QuickBASIC, Version 4.0 innerhalb von Include-Dateien nicht erlaubt. Wenn Sie Include-Dateien verwendet haben, um **SUB**-Prozeduren in Programmen zu definieren, die mit den QuickBASIC-Versionen 2.0 oder 3.0 geschrieben sind, müssen Sie diese Definitionen entweder in ein separates Modul schreiben, oder sie dort in das Modul einfügen, wo sie aufgerufen werden. (Weitere Informationen, wie Sie dies durchführen, finden Sie in Kapitel 4, "Wie Sie Quelldateien verwalten" in *Lernen und Anwenden von Microsoft QuickBASIC*.)

#### 2.5.4.4 Wie Sie Prozeduren in Quick-Bibliotheken deklarieren

Eine vorteilhafte Programmierpraxis ist es, alle Deklarationen von Prozeduren einer Quick-Bibliothek in eine Include-Datei zu schreiben. Mit dem Metabefehl **$INCLUDE** können Sie dann diese Include-Datei in Programme einfügen, die die Bibliothek verwenden. Dies bewahrt Sie vor dem Problem, alle notwendigen **DECLARE**-Anweisungen jedesmal zu kopieren, wenn Sie die Bibliothek verwenden.

### 2.5.5 Wie Sie Argumente als Referenz übergeben

Standardmäßig werden Variablen – ob einfache skalare Variablen, Datenfelder und Datenfeldelemente oder Verbunde – an **FUNCTION**- und **SUB**-Prozeduren "als Referenz" übergeben. Die folgende Liste erläutert, was mit "Variablenübergabe als Referenz" gemeint ist:

- Jede Programmvariable hat eine Adresse oder Stelle im Speicher, an der ihr Wert gespeichert ist.
- Der Aufruf einer Prozedur und die Übergabe von Variablen als Referenz an sie ruft die Prozedur auf und übergibt ihr die Adresse jeder Variablen. Daher sind die Adresse der Variablen und die Adresse des zugehörigen Parameters in der Prozedur ein und dasselbe.
- Deshalb ist auch der Wert einer an sie übergebenen Variablen verändert, wenn die Prozedur den Wert des Parameters verändert.

Wenn Sie nicht wollen, daß eine Prozedur den Wert einer Variablen verändert, übergeben Sie den in der Variablen enthaltenen Wert, nicht die Adresse. In diesem Fall werden Änderungen nur mit einer Kopie der Variablen, nicht mit der Variablen selbst, durchgeführt. Im nächsten Abschnitt finden Sie Erläuterungen zu dieser anderen Möglichkeit der Variablenübergabe.

**Beispiel**

Im folgenden Programm ändern Änderungen des Parameters `A$` in der Prozedur `Ersetze` ebenso das Argument `Test$`:

```
Test$ = "eine zeichenkette mit kleinbuchstaben."
PRINT "Vor Aufruf des Unterprogramms: "
PRINT Test$
CALL Ersetze (Test$,"t")
PRINT "Nach Aufruf des Unterprogramms: "
PRINT Test$
END
SUB Ersetze(A$, B$) STATIC
   Start = 1
   DO
      ' Suche nach B$ in A$, beginnend am Zeichen mit
      ' Position "Start" in A$:
      Gefunden = INSTR(Start, A$, B$)
      ' Wandele jedes Auftauchen von B$ in A$ in einen
      ' Großbuchstaben um:
      IF Gefunden > 0 THEN
          MID$(A$,Gefunden) = UCASE$(B$)
          Start = Start + 1
      END IF
   LOOP WHILE Gefunden > 0
END SUB
```

***Ausgabe***

```
Vor Aufruf des Unterprogramms:
eine zeichenkette mit kleinbuchstaben.
Nach Aufruf des Unterprogramms:
eine zeichenkeTTe mit kleinbuchsTaben.
```

## 2.5.6 Wie Sie Argumente als Wert übergeben

Die Übergabe eines Argumentes "als Wert" bedeutet, daß der Wert des Argumentes und nicht seine Adresse übergeben wird. In BASIC-Prozeduren wird die Übergabe einer Variablen als Wert simuliert, indem die Variable in eine temporäre Speicherstelle kopiert und dann die Adresse dieser temporären Speicherstelle übergeben wird. Da die Prozedur keinen Zugriff auf die Adresse der Ursprungsvariablen hat, kann sie die Ursprungsvariable nicht ändern. Stattdessen führt sie alle Änderungen mit der Kopie durch.

***Hinweis*** Wie bereits in Abschnitt 2.5.2, "Wie Sie Konstanten und Ausdrücke übergeben", hingewiesen wurde, können Sie Ausdrücke als Argumente an Prozeduren übergeben:

```
' A+B ist ein Ausdruck; dieser Prozeduraufruf hat
' keine Auswirkungen auf die Werte von A und B:
CALL Mult(A+B,B) STATIC
```

Ausdrücke werden immer als Wert übergeben.

### Beispiel

Eine Möglichkeit, eine Variable als Wert zu übergeben, ist, sie in Klammern einzuschließen und somit zu einem Ausdruck zu machen. Wie Sie in der folgenden Ausgabe sehen können, werden Änderungen der Variablen `Y`, die lokal zu der **SUB**-Prozedur ist, als Änderungen der Variablen `B` an den Modul-Ebenen-Code zurückgegeben. Änderungen von `X` in der Prozedur haben jedoch keine Auswirkung auf den Wert von `A`, da `A` als Wert übergeben wird.

```
A = 1
B = 1
PRINT "Vor Aufruf des Unterprogrammes, A ="; A; ", B ="; B
' A ist als Wert übergeben, B ist als Referenz
' übergeben:
CALL Mult((A), B)
PRINT "Nach Aufruf des Unterprogrammes, A ="; A; ", B ="; B
END
SUB Mult (X, Y) STATIC
   X = 2 * X
   Y = 3 * Y
   PRINT "Im Unterprogramm, X ="; X; ", Y ="; Y
END SUB
```

### *Ausgabe*

```
Vor Aufruf des Unterprogrammes, A = 1 , B = 1
Im Unterprogramm, X = 2 , Y = 3
Nach Aufruf des Unterprogrammes, A = 1 , B = 3
```

## 2.6 Die gemeinsame Nutzung von Variablen mit SHARED

Zusätzlich zur Übergabe von Variablen mit Argumenten- und Parameterlisten können Prozeduren auf eine der zwei unten gezeigten Arten Variablen mit anderen Prozeduren und mit Code auf Modul-Ebene (d.h. ist Code innerhalb eines Moduls, aber außerhalb jeder Prozedur) gemeinsam benutzen:

1. Variablen, die innerhalb einer Prozedur in einer **SHARED**-Anweisung aufgeführt sind, werden nur von dieser Prozedur und dem Modul-Ebenen-Code gemeinsam benutzt. Verwenden Sie diese Methode, wenn unterschiedliche Prozeduren im selben Modul verschiedene Kombinationen von Variablen der Modul-Ebene benötigen.
2. Variablen, die in einer **COMMON SHARED-**, **DIM SHARED-** oder **REDIM SHARED**-Anweisung auf Modul-Ebene aufgeführt sind, werden von dem Modul-Ebenen-Code und allen Prozeduren innerhalb dieses Moduls gemeinsam benutzt. Diese Methode ist besonders vorteilhaft, wenn alle Prozeduren eines Moduls einen gemeinsamen Satz von Variablen verwenden.

Damit zwei oder mehr Module Variablen gemeinsam benutzen können, können Sie auch die Anweisungen **COMMON** oder **COMMON SHARED** verwenden.

In Abschnitt 2.6.1 bis 2.6.3 werden diese drei Möglichkeiten, Variablen gemeinsam zu benutzen, erläutert.

### 2.6.1 Die gemeinsame Nutzung von Variablen mit bestimmten Prozeduren in einem Modul

Wenn unterschiedliche Prozeduren innerhalb eines Moduls unterschiedliche Variablen mit dem Modul-Ebenen-Code gemeinsam benutzen müssen, sollten Sie in jeder Prozedur die Anweisung **SHARED** verwenden.

Datenfelder in **SHARED**-Anweisungen bestehen aus dem Namen des Datenfeldes gefolgt von leeren Klammern ( ):

```
SUB AndereSub STATIC
   SHARED FeldName ()
   .
   .
   .
```

***Hinweis*** Wenn Sie mit einer **AS** Typ-Klausel einer Variablen ihren Typ zugewiesen haben, dann muß die **AS** Typ-Klausel in einer **SHARED**-Anweisung zusammen mit dem Namen der Variablen erscheinen:

```
DIM Puffer AS STRING * 10
.
.
.
END
SUB LiesSatz STATIC
   SHARED Puffer AS STRING * 10
   .
   .
   .
END SUB
```

### Beispiel

Im nächsten Beispiel zeigen die **SHARED**-Anweisungen in den Prozeduren `HolSatz` und `InventGes` das Format einer Liste von gemeinsam benutzten Variablen:

```
'=====================================================
'   MODUL-EBENEN-CODE
'=====================================================
TYPE SatzTyp
   Preis   AS SINGLE
   Beschr  AS STRING*35
END TYPE
DIM SatzVar(1 TO 100) AS SatzTyp   'Verbunddatenfeld
INPUT "Dateiname: ",DateiBeschr$
CALL HolSatz
PRINT InventGes
END
'=====================================================
'   PROZEDUR-EBENEN-CODE
'=====================================================
SUB HolSatz STATIC
'Sowohl DateiBeschr$ als auch das Verbunddatenfeld
'SatzVar werden mit dem obigen Modul-Ebenen-Code
'gemeinsam benutzt:
```

```
    SHARED DateiBeschr$, SatzVar() AS SatzTyp
    OPEN DateiBeschr$ FOR RANDOM AS #1
    .
    .
    .
END SUB
FUNCTION InventGes STATIC
'Nur das Datenfeld SatzVar wird mit dem Modul-Ebenen-
'Code gemeinsam benutzt:
    SHARED SatzVar() AS SatzTyp
    .
    .
    .
END FUNCTION
```

## 2.6.2 Die gemeinsame Nutzung von Variablen mit allen Prozeduren in einem Modul

Wenn Variablen auf Modul-Ebene mit dem Attribut **SHARED** in einer **COMMON**-, **DIM**- oder **REDIM**-Anweisung deklariert sind (zum Beispiel der Ausdruck **COMMON SHARED** *Variablenliste*), dann haben alle Prozeduren innerhalb dieses Moduls Zugriff auf diese Variablen; mit anderen Worten macht das Attribut **SHARED** Variablen global im gesamten Modul.

Das Attribut **SHARED** ist hilfreich, wenn Sie viele Variablen in allen Prozeduren eines Moduls gemeinsam benutzen müssen.

### Beispiel

Die folgenden Anweisungen deklarieren Variablen zur gemeinsamen Benutzung in allen Prozeduren eines einzelnen Moduls:

```
COMMON SHARED A, B, C
DIM SHARED Feld(1 TO 10, 1 TO 10) AS BenutzTyp
REDIM SHARED Alpha(N%)
```

In dem folgenden Beispiel werden das Zeichenkettendatenfeld `ZeichFeld` und die Ganzzahlvariablen `Min` und `Max` vom Modul-Ebenen-Code und den beiden **SUB**-Prozeduren `FuelleFeld` und `SchreibFeld` gemeinsam benutzt:

```
'=====================================================
'   MODUL-EBENEN-CODE
'=====================================================
DECLARE SUB FuelleFeld  ()
DECLARE SUB SchreibFeld ()
' Wegen der folgenden DIM-Anweisungen werden die
' Ganzzahlvariablen Min und Max sowie das
' Zeichenkettendatenfeld ZeichFeld von allen SUB oder
' FUNCTION in diesem Modul gemeinsam benutzt:
DIM SHARED ZeichFeld (33 TO 126) AS STRING * 5
DIM SHARED Min AS INTEGER, Max AS INTEGER
Min = LBOUND(ZeichFeld)
Max = UBOUND(ZeichFeld)
FuelleFeld   'Beachte die Abwesenheit von
             'Argumentenlisten.
SchreibFeld
END
'=====================================================
'      PROZEDUR-EBENEN-CODE
'=====================================================
SUB FuelleFeld STATIC
   ' Belege jedes Element des Datenfeldes von 33 bis 126
   ' mit einer Zeichenkette mit 5 Zeichen, von denen
   ' jedes Zeichen den ASCII-Code I% hat:
   FOR I% = Min TO Max
      ZeichFeld(I%) = STRING$(5,I%)
   NEXT
END SUB
SUB SchreibFeld STATIC
   FOR I% = Min TO Max
      PRINT ZeichFeld(I%)
   NEXT
END SUB
```

### Ein Teil der Ausgabe

```
!!!!!
"""""
#####
$$$$$
%%%%%
&&&&&
'''''
{{{{{
.
.
.
```

Wenn Sie zum Schreiben Ihrer Programme einen eigenen Texteditor verwenden und diese Programme direkt außerhalb der QuickBASIC-Programmentwicklungsumgebung kompilieren, müssen Sie beachten, daß Variablendeklarationen mit dem Attribut **SHARED** der Prozedurdefinition vorangehen müssen. Andernfalls ist der Wert keiner der mit **SHARED** deklarierten Variablen in der Prozedur verfügbar, wie in der folgenden Ausgabe des nächsten Beispiels gezeigt. (Wenn Sie QuickBASIC zum Erstellen Ihrer Programme verwenden, ist es nicht notwendig, diesem Ablauf zu folgen, da QuickBASIC Programme automatisch in der richtigen Reihenfolge speichert.)

```
DEFINT A-Z
FUNCTION Addiere (X,Y) STATIC
   Addiere = X + Y + Z
END FUNCTION
DIM SHARED Z
Z = 2
PRINT Addiere (1, 3)
END
```

### Ausgabe

```
 4
```

Das nächste Beispiel zeigt, wie Sie das oben abgebildete Modul speichern sollten. Benutzen Sie die Anweisung `DIM SHARED Z` gefolgt von der Definition von `Addiere`:

```
DEFINT A-Z
DECLARE FUNCTION Addiere (X,Y)
' Die Variable Z wird nun mit Addiere gemeinsam
' benutzt:
DIM SHARED Z
Z = 2
PRINT Addiere (1,3)
END
FUNCTION Addiere(X,Y) STATIC
   Addiere = X + Y + Z
END FUNCTION
```

***Ausgabe***

```
 6
```

## 2.6.3 Die gemeinsame Nutzung von Variablen mit anderen Modulen

Wenn Sie in Ihrem Programm Variablen zwischen Modulen gemeinsam benutzen wollen, listen Sie die Variablen in **COMMON**- oder **COMMON SHARED**-Anweisungen in jedem Modul auf der Modul-Ebene auf.

### Beispiele

Das folgende Beispiel zeigt, wie Variablen zwischen Modulen durch Verwendung von **COMMON** in dem Modul, das die **SUB**-Prozedur aufruft, und **COMMON SHARED** in dem Modul, das die Prozedur definiert, gemeinsam benutzt werden. Mit **COMMON SHARED** haben alle Prozeduren des zweiten Moduls Zugriff auf die gemeinsamen Variablen:

```
'=====================================================
'                        HAUPT-MODUL
'=====================================================
'
COMMON A, B
A = 2.5
B = 1.2
CALL Quadrat
```

```
CALL Kubikzahl
END
'=====================================================
'    Modul mit den Prozeduren Kubikzahl und Quadrat
'=====================================================
'
' HINWEIS: Die Namen der Variablen (X,Y) müssen nicht
' die gleichen sein wie in dem anderen Modul (A,B),
' müssen aber vom gleichen Typ sein.
'
COMMON SHARED X, Y  ' Diese Anweisung ist auf der
                    ' Modul-Ebene.
                    ' Sowohl X als auch Y werden mit
                    ' den nachfolgenden Prozeduren
                    ' KUBIKZAHL und QUADRAT gemeinsam
                    ' benutzt.
SUB Kubikzahl STATIC
   PRINT "A hoch 3 ="; X ^ 3
   PRINT "B hoch 3 ="; Y ^ 3
END SUB
SUB Quadrat STATIC
   PRINT "A quadrat ="; X ^ 2
   PRINT "B quadrat ="; Y ^ 2
END SUB
```

Das folgende Beispiel verwendet benannte **COMMON**-Blöcke auf den Modul-Ebenen und **SHARED**-Anweisungen innerhalb der Prozeduren, um unterschiedliche Sätze von Variablen mit jeder Prozedur gemeinsam zu benutzen (weitere Informationen zu benannten **COMMON**-Blöcken finden Sie in der Beschreibung der **COMMON**-Anweisung im *BASIC-Befehlsverzeichnis*):

```
' ====================================================
'                      HAUPT-MODUL
'   Gibt das Volumen und die Dichte eines gefüllten
'   Zylinders mit den eingegebenen Werten aus
'====================================================
'
COMMON /VolWerte/ Hoehe, Radius, Volumen
COMMON /DichtWerte/ Gewicht, Dichte
INPUT "Höhe des Zylinders in Zentimeter: ",Hoehe
INPUT "Radius des Zylinders in Zentimeter: ", Radius
INPUT "Gewicht des gefüllten Zylinders in Gramm: ", Gewicht
CALL VolBerech
CALL DichteBerech
```

```
PRINT "Volumen ist"; Volumen; "Kubikzentimeter."
PRINT "Dichte ist"; Dichte; "Gramm/Kubikzentimeter."
END
'=====================================================
' Modul mit den Prozeduren DichteBerech und VolBerech
'=====================================================
'
COMMON /VolWerte/ H, R, V
COMMON /DichtWerte/ G, D
SUB VolBerech STATIC
   ' Benutze die Variablen Hoehe, Radius und Volumen
   ' gemeinsam mit dieser Prozedur:
   SHARED H, R, V
   CONST PI = 3.141592653589#
   V = PI * H * (R ^ 2)
END SUB
SUB DichteBerech STATIC
   ' Benutze die Variablen Gewicht, Volumen und Dichte
   ' gemeinsam mit dieser Prozedur:
   SHARED G, V, D
   D = G / V
END SUB
```

***Ausgabe***

```
Höhe des Zylinders in Zentimeter: 100
Radius des Zylinders in Zentimeter: 10
Gewicht des gefüllten Zylinders in Gramm: 10000
Volumen ist 31415.93 Kubikzentimer.
Dichte ist 0.3183099 Gramm/Kubikzentimeter.
```

## 2.6.4 Das Problem von adreßgleichen Variablen

"Adreßgleiche Variablen" ist ein Problem, das manchmal in langen Programmen erscheint, die viele Variablen und Prozeduren enthalten. Adreßgleiche Variablen entstehen, wenn zwei oder mehr Namen sich auf denselben Platz im Speicher beziehen. Dies kann in den folgenden Situationen vorkommen:

- Wenn in der an eine Prozedur übergebenen Argumentenliste dieselbe Variable zweimal verwendet wird.
- Wenn von einer Prozedur auf eine Variable, die in einer Argumentenliste übergeben wird, mit Hilfe der **SHARED**-Anweisung oder des **SHARED**-Attributes zugegriffen wird.

Um aus Adreßgleichheit resultierende Probleme zu vermeiden, prüfen Sie Variablen, die mit einer Prozedur gemeinsam benutzt werden, doppelt um sicherzustellen, daß die Variablen nicht auch in der Argumentenliste eines Prozeduraufrufs erscheinen. Übergeben Sie dieselbe Variable ebenso nicht zweimal, wie in der folgenden Anweisung:

```
'X wird zweimal übergeben; dies wird in der Prozedur
'Test zu einem Problem adreßgleicher Variablen führen:
CALL Test (X, X, Y)
```

### Beispiele

Das folgende Beispiel verdeutlicht, wie adreßgleiche Variablen auftreten können. Hier wird die Variable A zwischen dem Modul-Ebenen-Code und der **SUB**-Prozedur gemeinsam benutzt mit der Anweisung `DIM SHARED`. A wird jedoch ebenso als Argument als Referenz an die **SUB** übergeben. Daher beziehen sich in dem Unterprogramm sowohl A als auch X auf denselben Platz im Speicher. Deshalb ändert das Unterprogramm, wenn es X ändert, auch A und umgekehrt.

```
DIM SHARED A
A = 4
CALL SchreibHaelfte(A)
END
SUB SchreibHaelfte (X) STATIC
   PRINT "Hälfte von"; X; "plus Hälfte von"; A;
   PRINT "ist    gleich";
   X = X / 2       ' X und A sind jetzt beide gleich 2.
   A = A / 2       ' X und A sind jetzt beide gleich 1.
   PRINT A + X
END SUB
```

#### *Ausgabe*

```
Hälfte von 4 plus Hälfte von 4 ist gleich 2
```

## 2.7 Automatische und STATIC-Variablen

Wenn das Attribut **STATIC** auf einer Prozedurdefinitionszeile erscheint, bedeutet dies, daß lokale Variablen innerhalb der Prozedur standardmäßig **STATIC** sind; das heißt, ihre Werte werden zwischen den Aufrufen der Prozedur erhalten.

Das Auslassen des **STATIC**-*Attributes* läßt lokale Variablen innerhalb der Prozedur standardmäßig "automatisch" werden; das heißt, Sie erhalten bei jedem Prozeduraufruf einen Satz lokaler Variablen mit neuen Werten.

Durch die Verwendung der **STATIC**-*Anweisung* innerhalb der Prozedur können Sie den Effekt des Auslassens des Attributes **STATIC** überschreiben, und damit einige Variablen automatisch und andere **STATIC** werden lassen (weitere Informationen finden Sie in Abschnitt 2.8).

***Hinweis*** Die Anweisung **SHARED** überschreibt ebenso die Standardeinstellung für Variablen in einer Prozedur (lokal **STATIC** oder lokal automatisch), da jede Variable, die in einer **SHARED**-Anweisung erscheint, auf Modul-Ebene definiert ist und daher nicht lokal zu der Prozedur ist.

## 2.8 Beibehaltung der Werte von lokalen Variablen mit der STATIC-Anweisung

Manchmal möchten Sie vielleicht einige lokale Variablen in einer Prozedur **STATIC** machen, während der Rest automatisch bleiben soll. Sie können dies erreichen, indem Sie diese Variablen innerhalb der Prozedur in einer **STATIC**-Anweisung aufführen.

Um sicher zu gehen, daß die Variable lokal ist, können Sie einen Variablennamen in eine **STATIC**-Anweisung schreiben, da eine **STATIC**-Anweisung selbst den Effekt einer **SHARED**-Anweisung auf Modul-Ebene überschreibt.

Eine **STATIC**-Anweisung kann nur innerhalb einer Prozedur erscheinen.

***Hinweis*** Datenfelder in **STATIC**-Anweisungen bestehen aus dem Datenfeldnamen gefolgt von einem Satz leerer Klammern ( ). Ebenso müssen Sie vor seiner Verwendung jedes Datenfeld, das in einer **STATIC**-Anweisung erscheint, dimensionieren, wie im nächsten Beispiel gezeigt:

```
SUB UnterProg2
   STATIC Datenfeld() AS INTEGER
   DIM Datenfeld(-5 TO 5, 1 TO 25) AS INTEGER
   .
   .
   .
END SUB
```

Beachten Sie in dem vorhergehenden Beispiel, wenn Sie einer Variablen ihren Typ in einer Klausel **AS** Typ zuweisen, daß dann die **AS** Typ-Klausel zusammen mit dem Namen der Variablen sowohl in der **STATIC**- als auch in der **DIM**-Anweisung erscheinen muß.

### Beispiel

Das folgende Beispiel zeigt, wie eine **STATIC**-Anweisung den Wert der Zeichenkettenvariablen `Y$` während folgender Aufrufe von `TestSub` beibehält:

```
DECLARE SUB TestSub ()
FOR I% = 1 TO 5
   TestSub              ' Rufe TestSub fünfmal auf.
NEXT I%
END
SUB TestSub             ' Beachte: kein STATIC-Attribut
   ' Sowohl X$ als auch Y$ sind in TestSub lokale
   ' Variablen (das bedeutet, ihre Werte werden nicht
   ' gemeinsam mit dem Modul-Ebenen-Code benutzt).
   ' Da X$ jedoch eine automatische Variable ist, wird
   ' sie bei jedem Aufruf von TestSub zu einer Null-
   ' Zeichenkette initialisiert. Im Gegensatz dazu
   ' ist Y$ STATIC, so daß sie den Wert behält, den sie
   ' vom letzten Aufruf hat:
   STATIC Y$
   X$ = X$ + "*"
   Y$ = Y$ + "*"
   PRINT X$, Y$
END SUB
```

### *Ausgabe*

```
*           *
*           **
*           ***
*           ****
*           *****
```

## 2.9 Rekursive Prozeduren

In BASIC können Prozeduren rekursiv sein. Eine rekursive Prozedur ist eine Prozedur, die sich selbst aufrufen kann oder andere Prozeduren aufrufen kann, die wieder die erste Prozedur aufrufen.

## 2.9.1 Die Funktion Fakultät

Eine gute Möglichkeit, eine rekursive Prozedur zu verdeutlichen, ist die Betrachtung der Funktion Fakultät aus der Mathematik. Eine Möglichkeit, n! ("n-Fakultät") zu definieren, stellt die folgende Formel dar:

```
n! = n*(n-1)*(n-2)*...*2*1
```

Zum Beispiel wird 5-Fakultät wie folgt ausgewertet:

```
5! = 5*4*3*2*1 = 120
```

***Hinweis*** Bringen Sie das in dieser Erläuterung verwendete mathematische Symbol für Fakultät nicht mit dem von QuickBASIC benutzten Typdeklarationssuffix für Zahlen einfacher Genauigkeit durcheinander.

Fakultäten eignen sich auch für eine rekursive Definition:

```
n! = n*(n-1)!
```

Dies führt zum folgenden Verlauf:

5! = 5*4!
4! = 4*3!
3! = 3*2!
2! = 2*1!
1! = 1*0!

Rekursion muß immer eine Abbruchbedingung aufweisen. Bei Fakultäten tritt diese Abbruchbedingung auf, wenn 0! ausgewertet wird - 0! ist per Definition gleich 1.

***Hinweis*** Obwohl eine rekursive Prozedur standardmäßig **STATIC**-Variablen enthalten kann (wie im nächsten Beispiel), ist es zumeist ratsam, stattdessen automatische Variablen zu verwenden. Mit automatischen Variablen als Standardeinstellung müssen Sie sich keine Gedanken darüber machen, daß rekursive Aufrufe einer Prozedur Variablenwerte eines vorhergehenden Aufrufs überschreiben.

### Beispiel

Das folgende Beispiel verwendet eine rekursive **FUNCTION**-Prozedur zur Berechnung von Fakultäten:

```
DECLARE FUNCTION Fakultaet# (N%)
Format$ = "###_! = ###################"
DO
   PRINT "Geben Sie eine Zahl von 0 bis 20 ein";
   INPUT "(oder - 1 zum Beenden): ", Zahl%
   IF Zahl% >= 0 AND Zahl% <= 20 THEN
      PRINT USING Format$; Zahl%; Fakultaet#(Zahl%)
   END IF
LOOP WHILE Zahl% >= 0
END
FUNCTION Fakultaet# (N%) STATIC
   IF N% > 0 THEN  'Rufe Fakultaet# erneut auf, wenn N
                   'größer als null ist.
      Fakultaet# = N% * Fakultaet#(N% - 1)
   ELSE            ' Ist das Ende der rekursiven
                   ' Aufrufe erreicht (N% = 0), so
                   ' "klettere die Leiter wieder
                   ' hinauf".
      Fakultaet# = 1
   END IF
END FUNCTION
```

## 2.9.2 Anpassen der Stapelgröße

Rekursion kann viel Speicherplatz benötigen, da jeder Satz von automatischen Variablen in einer **SUB**- oder **FUNCTION**-Prozedur auf dem Stapel (Stack) gespeichert wird. (Das Speichern von Variablen auf diese Art erlaubt es einer Prozedur, mit korrekten Variablenwerten weiterzuarbeiten, nachdem die Kontrolle von einem rekursiven Aufruf zurückgegeben ist.)

Wenn Sie eine rekursive Prozedur mit vielen automatischen Variablen oder eine tief verschachtelte rekursive Prozedur haben, müssen Sie eventuell die Größe des Stapels mit einer Anweisung **CLEAR** , , *Stapelgröße* anpassen, wobei *Stapelgröße* die Anzahl der zu reservierenden Bytes des Stapels ist. Andernfalls können Sie während des Programmlaufes die Fehlermeldung `Außerhalb des Stapelbereiches` erhalten.

Die folgenden Schritte zeigen eine Möglichkeit, die Größe des Speicherplatzes, den eine rekursive Prozedur benötigt, abzuschätzen:

1. Verwenden Sie die Funktion **FRE(-2)**, die den gesamten unbenutzten Stapelplatz angibt, zu Beginn und am Ende Ihres Programmes, um zu sehen, wieviel Stapelplatz Ihr Programm verwendet:

   Verwendete Bytes = **FRE(-2)**$_{Beginn}$ - **FRE(-2)**$_{Ende}$

   wobei **FRE(-2)**$_{Beginn}$ und **FRE(-2)**$_{Ende}$ die Werte darstellen, die von **FRE(-2)** zu Beginn und am Ende des Programmes ausgegeben werden.
2. Nachdem Sie die Logik Ihres Programmes in Betracht gezogen haben, fügen Sie einen "Sicherheitsfaktor" zu dem Betrag aus Schritt 1 hinzu, um den Wert *Gesamtbytes* zu erhalten, der in dem folgenden Schritt verwendet wird.
3. Reservieren Sie den in Schritt 2 berechneten Betrag an Stapelplatz:

   **CLEAR** , , *Gesamtbytes*

## 2.10 Übertragen der Kontrolle an ein anderes Programm mit CHAIN

Im Unterschied zu Prozeduraufrufen, die innerhalb desselben Programmes erscheinen, startet die Anweisung **CHAIN** einfach ein neues Programm. Wenn ein Programm zu einem anderen Programm verkettet, ergibt sich die folgende Reihenfolge:

1. Das erste Programm hält an.
2. Das zweite Programm wird in den Speicher geladen.
3. Das zweite Programm wird gestartet.

Der Vorteil bei der Verwendung von **CHAIN** besteht darin, daß Sie in der Lage sind, ein Programm mit großem Speicherbedarf in mehrere kleinere Programme aufzuteilen.

Die **COMMON**-Anweisung erlaubt es Ihnen, Variablen von einem Programm an ein anderes in der Kette zu übergeben. Es ist übliche Programmierpraxis, diese **COMMON**-Anweisungen in eine Include-Datei zu schreiben und anschließend den Metabefehl **$INCLUDE** am Kopf jedes Programmes in der Kette zu verwenden.

***Hinweis*** Verwenden Sie keine Anweisung **COMMON** */Blockname/ Variablenliste* (auch "benannter **COMMON**-Block" genannt), um Variablen an ein verkettetes Programm zu übergeben, da Variablen, die in einem benannten **COMMON**-Block aufgeführt sind, beim Verketten nicht erhalten bleiben. Verwenden Sie stattdessen unbenannte **COMMON**-Blöcke (**COMMON** *Variablenliste*). (Weitere Informationen zu benannten **COMMON**-Blöcken finden Sie in der Beschreibung zur Anweisung **COMMON** im *BASIC-Befehlsverzeichnis.*)

### Beispiel

Dieses Beispiel, das aus einer Kette besteht, die drei einzelne Programme miteinander verbindet, verwendet eine Include-Datei, um Variablen zu deklarieren, die von den Programmen gemeinsam benutzt werden.

```
' ======== INHALT DER INCLUDE-DATEI COMMONS.BI =======
DIM Werte(10)
COMMON Werte(), ZahlWerte
' ====================== HAUPT.BAS ==================
'
' Lies den Inhalt der COMMONS.BI-Datei ein:
' $INCLUDE: 'COMMONS.BI'
   ' Gib die Daten ein:
   PRINT "Geben Sie die Anzahl der Datenwerte ein";
   INPUT ":", ZahlWerte
   FOR I = 1 TO ZahlWerte
      Anfrage$ = "Wert ("+LTRIM$(STR$(I))+")? "
      PRINT Anfrage$;
      INPUT "",Werte(I)
   NEXT I
   ' Der Benutzer soll die durchzuführende Berechnung
   ' angeben:
   PRINT "Berechnung (1=Standardabweichung,2=Mittelwert)";
   INPUT " ? ", Wahl
   ' Verkette nun zu dem richtigen Programm:
   SELECT CASE Wahl
      CASE 1:          ' Standardabweichung
         CHAIN "STABW"
      CASE 2:          ' Mittelwert
         CHAIN "MITTEL"
   END SELECT
END
```

```
' ================== STABW.BAS ================
' Berechnet die Standardabweichung einer Stichprobe
' ============================================
'
' $INCLUDE: 'COMMONS.BI'
   Sum     = 0          ' Normale Summe
   SumQuad = 0          ' Summe der quadrierten Werte
   FOR I = 1 TO ZahlWerte
      Sum     = Sum     + Werte(I)
      SumQuad = SumQuad + Werte(I)^2
   NEXT I
   StdAbw = SQR (SumQuad/ZahlWerte - (Sum/ZahlWerte)^2)
   PRINT "Die Standardabweichung der ";
   PRINT "Stichprobe ist: " StdAbw
END
' ================== MITTEL.BAS ================
'  Berechnet den Mittelwert (Durchschnitt)
'  einer Stichprobe
' ============================================
'
' $INCLUDE: 'COMMONS.BI'
   Sum = 0
   FOR I = 1 TO ZahlWerte
      Sum = Sum + Werte(I)
   NEXT
   Mittel = Sum/ZahlWerte
   PRINT "Der Mittelwert der Stichprobe ist: " Mittel
END
```

## 2.11 Anwendungsbeispiel: Rekursive Verzeichnisdurchsuchung (WOIST.BAS)

Das folgende Programm verwendet eine rekursive **SUB**-Prozedur, um eine Diskette/Festplatte nach dem Dateinamen zu durchsuchen, der vom Benutzer eingegeben wurde. Jedesmal, wenn dieses Programm die eingegebene Datei findet, gibt es den kompletten Pfad zu der Datei aus.

### Verwendete Anweisungen und Funktionen

Dieses Programm demonstriert die in diesem Kapitel erläuterten folgenden Anweisungen und Schlüsselworte:

- **DECLARE**
- **FUNCTION...END FUNCTION**
- **STATIC**
- **SUB...END SUB**

### Programmlisting

```
DEFINT A-Z
' Deklariere die im Programm verwendeten symbolischen
' Konstanten:
CONST EOFTYP = 0, DATEITYP = 1, DIRTYP = 2, ROOT = "TWH"
DECLARE SUB SuchVerz (Pfad$, Ebene, DateiAng$,    Zeile)
DECLARE FUNCTION ErstDateiNam$ (Zahl)
DECLARE FUNCTION HolEintr$ (DateiZahl, EintrTyp)
CLS
INPUT "Zu suchende Datei"; DateiAng$
PRINT
PRINT "Geben Sie das Verzeichnis ein, ";
PRINT "ab dem gesucht werden soll"
PRINT "(optional Laufwerk + Verzeichnisse)."
PRINT "Betätigen Sie die Eingabetaste, um die Suche im ";
PRINT "Hauptverzeichnis des aktuellen Laufwerks zu "
PRINT "beginnen."
PRINT
INPUT "Startverzeichnis"; Pfad$
CLS
RechtsZeich$ = RIGHT$(Pfad$, 1)
IF Pfad$ = "" OR RechtsZeich$ = ":" OR _
   RechtsZeich$ <>   "\" THEN
   Pfad$ = Pfad$ + "\"
END IF
DateiAng$ = UCASE$(DateiAng$)
Pfad$ = UCASE$(Pfad$)
Ebene = 1
Zeile = 3
```

```
' Führe Aufruf mit oberster Ebene (Ebene 1) durch,
' um die Suche zu beginnen:
SuchVerz Pfad$, Ebene, DateiAng$, Zeile

KILL ROOT + ".*"    ' Lösche alle vom Programm
                    ' angelegten temporären Dateien.
LOCATE Zeile + 1, 1: PRINT "Suche beendet."
END
'
' ================== HOLEINTR ======================
'  Diese Prozedur verarbeitet Eintragszeilen einer
'  DIR-Liste, die in einer Datei gespeichert ist.
'
'  Diese Prozedur gibt die folgenden Werte aus:
'
'  HolEintr$   Ein gültiger Datei- oder
'              Verzeichnisname
'  EintrTyp    Wenn gleich 1, ist HolEintr$ eine
'              Datei
'              Wenn gleich 2, ist HolEintr$ ein
'              Verzeichnis.
'=====================================================
'
FUNCTION HolEintr$ (DateiZahl, EintrTyp) STATIC

   ' Wiederhole, bis ein gültiger Eintrag oder Ende der
   ' Datei (EOF) gelesen ist:
   DO UNTIL EOF(DateiZahl)
      LINE INPUT #DateiZahl, EintrZeile$
      IF EintrZeile$ <> "" THEN

      ' Hole erstes Zeichen von der Zeile zur
      ' Überprüfung:
      TestZeich$ = LEFT$(EintrZeile$, 1)
     IF TestZeich$ <> " " AND TestZeich$ <> "." THEN EXIT DO
      END IF
   LOOP

   ' Entscheide, ob Eintrag oder EOF gefunden:
   IF EOF(DateiZahl) THEN      ' EOF, also gib EOFTYP
      EintrTyp = EOFTYP        ' in EintrTyp zurück.
      HolEintr$ = ""

   ELSE      ' Nicht EOF, also muß es eine Datei oder
             ' ein Verzeichnis sein.

     'Erstelle den Namen des Eintrags und gib ihn aus:
      EintrName$ = RTRIM$(LEFT$(EintrZeile$, 8))
```

```
      'Überprüfe auf Erweiterung und füge diese dem
      'Namen hinzu, sofern eine vorhanden ist:
       EintrErw$ = RTRIM$(MID$(EintrZeile$, 10, 3))
       IF EintrErw$ <> "" THEN
          HolEintr$ = EintrName$ + "." + EintrErw$
       ELSE
          HolEintr$ = EintrName$
       END IF
       'Bestimme den Typ des Eintrags und gib diesen
       'Wert zurück an die Stelle, an der HolEintr$
       'aufgerufen wurde.
       IF MID$(EintrZeile$, 15, 3) = "DIR" THEN
          EintrTyp = DIRTYP            ' Verzeichnis
       ELSE
          EintrTyp = DATEITYP          ' Datei
       END IF
   END IF
END FUNCTION
'
' ==================ERSTDATEINAM$====================
'  Diese Prozedur erstellt einen Dateinamen aus
'  einer Basis-Zeichenkette
'  ("TWH", als symbolische Konstante auf Modul-
'  Ebene definiert) und einer an sie als Argument
'  übergebenen Zahl (Zahl).
'=====================================================
'
FUNCTION ErstDateiNam$ (Zahl) STATIC
   ErstDateiNam$ = ROOT + "." + LTRIM$(STR$(Zahl))
END FUNCTION
'
' ================== SUCHVERZ ======================
'Diese Prozedur durchsucht rekursiv ein Verzeichnis
'nach dem Dateinamen, der vom Benutzer eingegeben
'wurde.
'
'BEACHTE: Der SUB-Kopf verwendet das Schlüsselwort
'         STATIC nicht, da diese Prozedur bei jedem
'         Aufruf einen neuen Satz von Variablen
'         benötigt.
'=====================================================
'
```

```
SUB SuchVerz (Pfad$, Ebene, DateiAng$, Zeile)
   LOCATE 1, 1: PRINT "Ich durchsuche jetzt"; SPACE$(50);
   LOCATE 1, 15: PRINT Pfad$;
   ' Erstelle eine Dateibeschreibung für die temporäre
   ' Datei:
   TempBeschr$ = ErstDateiNam$(Ebene)
   ' Hole eine Verzeichnisliste des aktuellen
   ' Verzeichnisses und speichere sie in der
   ' temporären Datei:
   SHELL "DIR " + Pfad$ + " > " + TempBeschr$
   ' Hole die nächste verfügbare Dateinummer:
   DateiZahl = FREEFILE
   ' Öffne die DIR-Listendatei und durchsuche sie:
   OPEN TempBeschr$ FOR INPUT AS #DateiZahl
   ' Verarbeite die Datei Zeile für Zeile:
   DO
      'Lies den Eintrag aus der DIR-Listendatei ein:
      DirEintr$ = HolEintr$(DateiZahl, EintrTyp)
      ' Wenn Eintrag eine Datei ist:
      IF EintrTyp = DATEITYP THEN
      ' Wenn die Zeichenkette DateiAng$ passend ist,
      ' gibt den Eintrag aus und verlasse diese
      ' Schleife:
      IF DirEintr$ = DateiAng$ THEN
         LOCATE Zeile, 1: PRINT Pfad$; DirEintr$;
         Zeile = Zeile + 1
         EintrTyp = EOFTYP
      END IF
      ' Wenn der Eintrag ein Verzeichnis ist, führe
      ' einen rekursiven Aufruf von SuchVerz mit dem
      ' neuen Verzeichnis durch:
      ELSEIF EintrTyp = DIRTYP THEN
         NeuPfad$ = Pfad$ + DirEintr$ + "\"
         SuchVerz NeuPfad$, Ebene + 1, DateiAng$, Zeile
         LOCATE 1,1:PRINT "Ich durchsuche jetzt";SPACE$(50);
         LOCATE 1, 15: PRINT Pfad$;
      END IF
   LOOP UNTIL EintrTyp = EOFTYP
   ' Die Suche in dieser Datei ist beendet, also
   ' schließe sie:
   CLOSE DateiZahl
END SUB
```

## 2.12 Zusammenfassung der Anweisungen, die in BASIC-Prozeduren verwendet werden

Die folgende Tabelle führt die Anweisungen auf, die in BASIC verwendet werden, um BASIC-Prozeduren zu definieren, zu deklarieren, aufzurufen und Argumente an sie zu übergeben. Die folgende Tabelle führt ebenso die Anweisungen auf, die verwendet werden, um Variablen zwischen Prozeduren, Modulen und separaten Programmen in einer Kette gemeinsam zu benutzen.

| *Aufgabe* | *Anweisung* | *Funktion* |
|---|---|---|
| Eine Prozedur definieren | **FUNCTION... END FUNCTION** | Markiert den Beginn bzw. das Ende einer **FUNCTION**-Prozedur. |
| | **SUB...END SUB** | Markiert den Beginn bzw. das Ende einer **SUB**-Prozedur. |
| Eine Prozedur aufrufen | **CALL** | Übergibt die Kontrolle an eine BASIC-**SUB**-Prozedur oder an eine Prozedur, die in einer anderen Programmiersprache geschrieben und separat kompiliert ist. (Das Schlüsselwort **CALL** ist wahlfrei.) |
| Eine Prozedur verlassen | **EXIT FUNCTION** | Bietet eine weitere Möglichkeit, eine **FUNCTION**-Prozedur zu verlassen. |
| | **EXIT SUB** | Bietet eine weitere Möglichkeit, eine **SUB**-Prozedur zu verlassen. |
| Auf eine Prozedur bezugnehmen, bevor sie definiert ist | **DECLARE** | Deklariert eine **FUNCTION** oder **SUB** und gibt optional die Anzahl und Typen ihrer Parameter an. |

| *Aufgabe* | *Anweisung* | *Funktion* |
|---|---|---|
| Variablen gemeinsam zwischen Modulen, Prozeduren oder Programmen benutzen | **COMMON** | Benutzt Variablen gemeinsam zwischen einzelnen Modulen. Bei der gleichzeitigen Verwendung des Attributs **SHARED** werden Variablen zwischen unterschiedlichen Prozeduren des gleichen Moduls gemeinsam benutzt. Übergibt ebenso Variablenwerte vom aktuellen Programm in ein neues Programm, wenn die Kontrolle mit der Anweisung **CHAIN** übertragen wird. |
| | **SHARED** | Wenn mit den Anweisungen **COMMON**, **DIM** oder **REDIM** auf Modul-Ebene (zum Beispiel **DIM SHARED**) verwendet, werden Variablen von jeder **SUB** oder **FUNCTION** in einem einzelnen Modul gemeinsam benutzt.<br>Wenn es allein innerhalb einer Prozedur verwendet wird, werden Variablen zwischen dieser Prozedur und dem Modul-Ebenen-Code gemeinsam benutzt. |
| Variablenwerte beibehalten | **STATIC** | Zwingt Variablen, lokal zu einer Prozedur oder **DEF FN**-Funktion zu sein, und behält die Werte bei, die in den Variablen gespeichert sind, wenn die Prozedur oder Funktion verlassen und anschließend wieder aufgerufen wird. |

| *Aufgabe* | *Anweisung* | *Funktion* |
|---|---|---|
| Eine mehrzeilige Funktion definieren | **DEF FN**<br>**END DEF** | Markiert den Beginn und das Ende einer mehrzeiligen **DEF FN**-Funktion (dies ist die alte Version für Funktionen in BASIC, **FUNCTION**-Prozeduren bieten eine leistungsfähige Alternative). |
| Eine mehrzeilige Funktion verlassen | **EXIT DEF** | Bietet einen weiteren Weg, eine mehrzeilige **DEF FN**-Funktion zu verlassen. |
| Eine BASIC-Unterroutine aufrufen | **GOSUB** | Übergibt die Kontrolle an eine spezielle Zeile in einem Modul. Die Kontrolle wird von der Unterroutine mit einer **RETURN**-Anweisung an die Zeile zurückgegeben, die der **GOSUB**-Anweisung folgt (dies ist die alte Version für Unterroutinen in BASIC, **SUB**-Prozeduren bieten eine leistungsfähige Alternative). |
| Kontrolle an ein anderes Programm übertragen | **CHAIN** | Überträgt die Kontrolle vom aktuellen Programm im Speicher an ein anderes Programm; verwenden Sie **COMMON**, um Variablen an das neue Programm zu übergeben. |

# 3 Datei- und Geräte-E/A


3.1 Wie Sie Text auf den Bildschirm ausgeben 3.3
- 3.1.1 Bildschirmzeilen und -spalten 3.4
- 3.1.2 Wie Sie Text und Zahlen mit **PRINT** anzeigen 3.5
- 3.1.3 Wie Sie formatierte Ausgaben mit **PRINT USING** anzeigen 3.6
- 3.1.4 Überspringen von Leerzeichen und Springen in eine angegebene Spalte 3.7
- 3.1.5 Wie Sie die Zeilen- und Spaltenzahl verändern 3.8
- 3.1.6 Wie Sie ein Text-Darstellungsfeld erstellen 3.8

3.2 Wie Sie Eingaben von der Tastatur einlesen 3.10
- 3.2.1 Die **INPUT**-Anweisung 3.11
- 3.2.2 Die **LINE INPUT**-Anweisung 3.12
- 3.2.3 Die **INPUT$**-Funktion 3.14
- 3.2.4 Die **INKEY$**-Funktion 3.14

3.3 Wie Sie den Text-Cursor steuern 3.15
- 3.3.1 Wie Sie den Cursor positionieren 3.16
- 3.3.2 Wie Sie die Cursor-Größe verändern 3.17
- 3.3.3 Wie Sie Informationen über die Position des Cursors erhalten 3.18

3.4 Wie Sie mit Datendateien arbeiten 3.19
- 3.4.1 Wie Datendateien organisiert sind 3.19
- 3.4.2 Sequentielle und Direktzugriffsdateien 3.20
- 3.4.3 Wie Sie eine Datendatei öffnen 3.20
  - 3.4.3.1 Dateinummern in BASIC 3.22
  - 3.4.3.2 Dateinamen in BASIC 3.22
- 3.4.4 Wie Sie Datendateien schließen 3.24

3.4.5 Wie Sie sequentielle Dateien verwenden 3.24
3.4.5.1 Sätze in sequentiellen Dateien 3.24
3.4.5.2 Wie Sie einer neuen sequentiellen Datei Daten hinzufügen 3.26
3.4.5.3 Wie Sie Daten aus einer existierenden sequentiellen Datei lesen 3.27
3.4.5.4 Wie Sie Daten an eine existierende sequentielle Datei anfügen 3.28
3.4.5.5 Weitere Möglichkeiten, Daten in eine sequentielle Datei zu schreiben 3.28
3.4.5.6 Weitere Möglichkeiten, Daten aus einer sequentiellen Datei zu lesen 3.31
3.4.6 Wie Sie Direktzugriffsdateien benutzen 3.33
3.4.6.1 Sätze in Direktzugriffsdateien 3.34
3.4.6.2 Wie Sie einer Direktzugriffsdatei Daten hinzufügen 3.35
3.4.6.3 Wie Sie Daten sequentiell aus einer Direktzugriffsdatei lesen 3.40
3.4.6.4 Wie Sie Satznummern einsetzen, um auf bestimmte Sätze einer Direktzugriffsdatei zuzugreifen 3.41
3.4.7 Binärdatei-E/A 3.43
3.4.7.1 Vergleich zwischen Binärzugriff und Direktzugriff 3.43
3.4.7.2 Wie Sie den Dateizeiger mit **SEEK** positionieren 3.44

3.5 Wie Sie mit Geräten arbeiten 3.47
3.5.1 Unterschiede zwischen Geräte- und Datei-E/A 3.48
3.5.2 Datenübertragung über einen seriellen Anschluß 3.50

3.6 Anwendungsbeispiele 3.51
3.6.1 Immerwährender Kalender (KAL.BAS) 3.51
3.6.2 Wie Sie eine Direktzugriffsdatei indizieren (INDEX.BAS) 3.56
3.6.3 Terminal-Emulator (TERMINAL.BAS) 3.64

3.7 Zusammenfassung der Standard-E/A-Anweisungen 3.67

3.8 Zusammenfassung der Datei-E/A-Anweisungen 3.68

Dieses Kapitel zeigt Ihnen, wie Sie die BASIC Eingabe- und Ausgabe (E/A) Funktionen und -Anweisungen verwenden, um Ihrem Programm die Fähigkeit zu verleihen, nicht nur auf in Dateien gespeicherte Daten zuzugreifen, sondern ebenso Daten mit Geräten auszutauschen, die an Ihr System angeschlossen sind.

Das Thema E/A umfaßt vielfältige Programmieraufgaben, die das Wiederfinden, das Speichern und Formatieren von Informationen einbeziehen. Dieses Thema schließt ebenso die Beziehung zwischen Datendateien und physikalischen Geräten, wie z.B. Bildschirm und Tastatur, ein.

Wenn Sie dieses Kapitel gelesen haben, werden Sie wissen, wie folgende Programmieraufgaben gelöst werden:

- Ausgabe von Text auf den Bildschirm.
- Eingaben von der Tastatur lesen, zur Verwendung im Programm.
- Datendateien auf Diskette anlegen.
- Speichern von Datensätzen in Datendateien.
- Lesen von Datensätzen aus Datendateien.
- Lesen oder Verändern von Daten in Dateien, die nicht auf dem American Standard Code for Information Interchange (ASCII) basieren.
- Daten mit anderen Computern über die serielle Schnittstelle austauschen.

## 3.1 Wie Sie Text auf den Bildschirm ausgeben

Dieser Abschnitt erläutert, wie die folgenden Aufgaben gelöst werden:

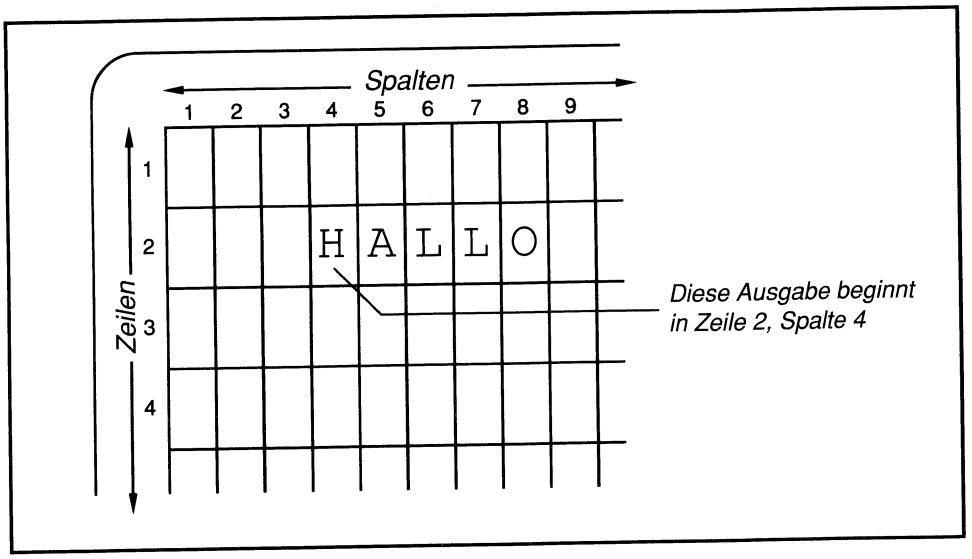
- Ausgabe von Text auf den Bildschirm mit **PRINT**.
- Ausgabe von formatiertem Text auf den Bildschirm mit **PRINT USING**.
- Überspringen von Leerzeichen in einer Ausgabezeile mit **SPC**.
- Springen in eine gegebene Spalte in einer Ausgabezeile mit **TAB**.
- Verändern der Spalten- und Zeilenanzahl auf dem Bildschirm mit **WIDTH**.
- Öffnen eines Text-Darstellungsfeldes mit **VIEW PRINT**.

***Hinweis*** Ausgabe, die auf dem Bildschirm erscheint, wird manchmal als "Standardausgabe" bezeichnet. Sie können die Standardausgabe mit den Symbolen > oder >> auf der DOS-Befehlszeile umleiten, so daß Ausgaben, die eigentlich auf den Bildschirm geleitet würden, zu einem anderen Ausgabegerät (z. B. einem Drucker) oder zu einer Diskettendatei gesendet werden. (Nähere Informationen zur Umleitung der Eingabe finden Sie in Ihrer DOS-Dokumentation.)

### 3.1.1 Bildschirmzeilen und -spalten

Um zu verstehen, wie Text auf den Bildschirm ausgegeben wird, ist es hilfreich, sich den Bildschirm als Raster von "Zeilen" und "Spalten" vorzustellen. Die Höhe einer Zeile überschreitet leicht die Höhe eines geschriebenen Zeichens, während die Breite einer Spalte etwas größer als die Breite eines Zeichens ist. Eine Standard-Bildschirmkonfiguration im Textmodus (nicht grafisch) hat eine Breite von 80 Spalten bei einer Höhe von 25 Zeilen. Abbildung 3.1 zeigt, wie jedes auf den Bildschirm ausgegebene Zeichen ein einziges Element des Rasters einnimmt, ein Element, das durch die Verbindung eines Zeilenargumentes mit einem Spaltenargument festgelegt werden kann.

*Abbildung 3.1 Textausgabe auf den Bildschirm*




Die unterste Zeile des Bildschirms wird normalerweise nicht für Ausgaben verwendet, solange Sie nicht eine **LOCATE**-Anweisung benutzen, um dort Text auszugeben. (Weitere Informationen zu **LOCATE** finden Sie in Abschnitt 3.3, "Wie Sie den Text-Cursor steuern".)

## 3.1.2 Wie Sie Text und Zahlen mit PRINT anzeigen

Die am häufigsten verwendete Anweisung für Ausgaben auf den Bildschirm ist die Anweisung **PRINT**. Mit **PRINT** können Sie numerische oder Zeichenkettenwerte oder eine Kombination aus beidem anzeigen. Außerdem gibt **PRINT** ohne Argumente eine leere Zeile aus.

Es folgen einige allgemeine Erläuterungen zu **PRINT**:

- **PRINT** gibt Zahlen immer mit einem nachfolgenden Leerzeichen aus. Wenn die Zahl positiv ist, geht ihr auch immer ein Leerzeichen voraus; wenn die Zahl negativ ist, geht ihr ein Minuszeichen voraus. (Weitere Informationen darüber, wie **PRINT** Werte einfacher und doppelter Genauigkeit behandelt, finden Sie in der Beschreibung für die **PRINT**-Anweisung im *BASIC-Befehlsverzeichnis*.)
- **PRINT** kann dazu verwendet werden, Listen von Ausdrücken auszugeben. Ausdrücke der Liste können durch Kommata, Semikolons, einem oder mehreren Leerzeichen oder einem oder mehreren Tabulatorzeichen von anderen Ausdrücken getrennt werden. Ein Komma veranlaßt **PRINT**, auf dem Bildschirm zum Anfang der nächsten "Ausgabezone", also zum Anfang des nächsten Blockes von 14 Spalten zu springen. Ein Semikolon (oder eine Kombination von Leerzeichen und/oder Tabs) zwischen zwei Ausdrücken schreibt die Ausdrücke auf den Bildschirm direkt nebeneinander ohne Leerzeichen (abgesehen von den "eingebauten" Leerzeichen für Zahlen).
- Normalerweise beendet **PRINT** jede Ausgabezeile mit einer Neue-Zeile (Wagenrücklauf-Zeilenvorschub) Sequenz . Ein Komma oder ein Semikolon am Ende der Ausdrucksliste unterdrückt dies jedoch; solange die nächste geschriebene Ausgabe nicht zu lang ist, um auf diese Zeile zu passen, erscheint die Ausgabe auf derselben Zeile.
- **PRINT** bricht eine Ausgabezeile, die länger als die Breite des Bildschirms ist, auf die nächste Zeile um. Wenn Sie zum Beispiel versuchen, eine Zeile, die 100 Zeichen lang ist, auf einen Bildschirm mit 80 Spalten auszugeben, werden die ersten 80 Zeichen der Zeile in eine Zeile geschrieben gefolgt von den nächsten 20 Zeichen auf der nächsten Zeile. Wenn die Zeile mit 100 Zeichen nicht in der linken Ecke des Bildschirms anfing (wenn sie zum Beispiel einer mit einem Komma oder mit einem Semikolon beendeten **PRINT**-Anweisung folgte), dann würde die Zeile bis zur 80. Spalte einer Zeile geschrieben werden und in der ersten Spalte der nächsten Zeile beginnend fortgesetzt.

**Beispiel**

Die Ausgabe des folgenden Programmes zeigt einige der unterschiedlichen Möglichkeiten, wie Sie **PRINT** benutzen können:

```
A =  2
B = -1
C =  3
X$ = "da"
Y$ = "hinten"
PRINT A, B, C
PRINT B, A, C
PRINT A; B; C
PRINT X$; Y$
PRINT X$, Y$;
PRINT A, B
PRINT
FOR I = 1 TO 8
   PRINT X$,
NEXT
```

***Ausgabe***

```
 2            -1            3
-1             2            3
 2 -1  3
dahinten
da           hinten 2     -1

da            da           da            da            da
da            da           da
```

## 3.1.3 Wie Sie formatierte Ausgaben mit PRINT USING anzeigen

Die **PRINT USING**-Anweisung ermöglicht im Unterschied zu **PRINT** eine bessere Kontrolle über das Erscheinungsbild geschriebener Daten, im besonderen numerischer Daten. Mit der Verwendung besonderer, in einer Formatzeichenkette eingebundener Zeichen, erlaubt **PRINT USING** es Ihnen, Informationen anzugeben, wie z. B. wieviele Ziffern einer Zahl oder (wieviele Zeichen einer Zeichenkette) angezeigt werden sollen, ob ein Pluszeichen (+) oder ein Dollarzeichen ($) vor einer Zahl erscheinen soll oder nicht usw. Eine komplette Liste dieser besonderen Zeichen finden Sie in der Beschreibung der **PRINT USING**-Anweisung im *BASIC-Befehlsverzeichnis.*

**Beispiel**

Das folgende Beispiel zeigt ein Muster dessen, was mit **PRINT USING** erreicht werden kann. Beachten Sie, daß Sie hinter der Formatzeichenkette von **PRINT USING** mehr als einen Ausdruck aufführen können. Identisch zu **PRINT** können die Anweisungen in der Liste voneinander durch Kommata, Semikolons, Leerzeichen oder Tabzeichen getrennt sein.

```
X = 441.2318
PRINT USING "Die Zahl mit 3 Dezimalstellen  ###.###";X
PRINT USING "Die Zahl mit einem Dollarzeichen  $$##.##";X
PRINT USING "Die Zahl im Exponentialformat  #.###^^^^";X
PRINT USING "Die Zahl mit Pluszeichen  +###"; X; 99.9
```

***Ausgabe***

```
Die Zahl mit 3 Dezimalstellen 441.232
Die Zahl mit einem Dollarzeichen $441.23
Die Zahl im Exponentialformat 0.441E+03
Die Zahl mit Pluszeichen +441  Die Zahl mit Pluszeichen +100
```

### 3.1.4 Überspringen von Leerzeichen und Springen in eine angegebene Spalte

Mit der Verwendung der Anweisung **SPC**(*n*) in einer **PRINT**-Anweisung können Sie *n* Leerzeichen in einer Ausgabezeile überspringen, wie mit der Ausgabe des nächsten Beispieles gezeigt wird:

```
PRINT "         1         2         3"
PRINT "12345678901234567890123456789O"
PRINT "Vorname"; SPC (10); "Nachname"
```

***Ausgabe***

```
         1         2         3
12345678901234567890123456789O
Vorname          Nachname
```

Mit der Verwendung der **TAB**(*n*)-Anweisung in einer **PRINT**-Anweisung können Sie in die *n*-te Spalte (fortfahrend von der linken Seite des Bildschirms) einer Zeile geschriebener Ausgabe springen. Das folgende Beispiel verwendet **TAB**, um dieselbe Ausgabe wie die des oberen Beispiels zu erstellen.

```
PRINT "         1         2         3"
PRINT "123456789012345678901234567890"
PRINT "Vorname"; TAB(18); "Nachname"
```

Weder **SPC** noch **TAB** können selbständig verwendet werden, um geschriebene Ausgabe auf dem Bildschirm zu positionieren; sie müssen in einer **PRINT**-Anweisung erscheinen.

## 3.1.5 Wie Sie die Zeilen- und Spaltenzahl verändern

Sie können die maximale Anzahl von Zeichen, die in einer einzigen Ausgabezeile erscheinen, mit der Anweisung **WIDTH** *Spalten* steuern. Die Anweisung **WIDTH** *Spalten* verändert die Größe der Zeichen, die auf den Bildschirm ausgegeben werden, so daß mehr bzw. weniger Zeichen in eine Zeile passen können. `WIDTH 40` zum Beispiel verbreitert die Zeichen, so daß die maximale Zeilenlänge 40 Zeichen ist. `WIDTH 80` verkleinert Zeichen, so daß die maximale Zeilenlänge 80 Zeichen ist. Die Zahlen 40 und 80 sind die einzig gültigen Werte für das Argument *Spalten*.

Auf Maschinen, die mit einem "Enhanced Graphics Adapter" (EGA) oder "Video Graphics Adapter" (VGA) ausgerüstet sind, kann die Anweisung **WIDTH** ebenso die Anzahl der auf dem Bildschirm erscheinenden Zeilen steuern, wie mit der folgenden Syntax gezeigt:

**WIDTH** [*Spalten*] [, *Zeilen*]

Der Wert für *Zeilen* kann 25, 30, 43, 50 oder 60 sein. Dies ist abhängig von dem Bildschirmadapter, den Sie verwenden, und dem in einer vorherigen **SCREEN**-Anweisung gesetzten Bildschirmmodus. (Weitere Informationen zu gültigen Werten für *Zeilen* in jedem Bildschirmmodus finden Sie in der Beschreibung für die Anweisung **SCREEN** im *BASIC-Befehlsverzeichnis*.)

## 3.1.6 Wie Sie ein Text-Darstellungsfeld erstellen

Bisher wurde der gesamte Bildschirm für Textausgaben verwendet. Mit der Anweisung **VIEW PRINT** [*oberste Zeile* **TO** *unterste Zeile*] können Sie jedoch geschriebene Ausgabe auf ein "Text-Darstellungsfeld" begrenzen. Unter einem "Text-Darstellungsfeld" versteht man einen horizontalen Ausschnitt des Bildschirms zwischen *oberste Zeile* und *unterste Zeile*.

Ein Text-Darstellungsfeld ermöglicht Ihnen ebenfalls Kontrolle über das Rollen (scrolling) auf dem Bildschirm. Ohne ein Text-Darstellungsfeld beginnt die Ausgabe zu rollen, wenn sie das untere Ende des Bildschirms erreicht hat. Sowohl Text als auch Graphik, die sich auf der jeweilig obersten Zeile des Bildschirms befinden, gehen beim Rollen des Bildschirms verloren. Nach einer **VIEW PRINT**-Anweisung jedoch findet Rollen nur noch zwischen der obersten und untersten Zeile des Darstellungsfeldes statt. Um nur das Text-Darstellungsfeld zu löschen, können Sie außerdem die Anweisung **CLS 2** benutzen, wobei der Rest des Bildschirminhaltes unberührt bleibt.

***Hinweis*** Wie ein graphisches Darstellungsfeld für graphische Ausgaben auf dem Bildschirm erstellt wird, wird in Abschnitt 5.5 beschrieben.

### Beispiel

Die Auswirkungen einer **VIEW PRINT**-Anweisung können Sie sehen, indem Sie die Ausgabe des nächsten Beispiels studieren.

```
CLS
LOCATE 3, 1
PRINT "Oberhalb des Text-Darstellungsfeldes, ";
PRINT "es wird aber nicht gerollt."
LOCATE 4, 1
PRINT STRING$(60, "_")     ' Schreibe horizontale
LOCATE 11, 1               ' Linien oberhalb und
PRINT STRING$(60, "_")     ' unterhalb des Text-
                           ' Darstellungsfeldes.
PRINT "Unterhalb des Text-Darstellungsfeldes."
VIEW PRINT 5 TO 10    ' Text-Darstellungsfeld erstreckt
                      ' sich von Zeile 5 bis 10.
FOR I = 1 TO 20       ' Schreibe Text und Zahlen in das
                      ' Darstellungsfeld.
   PRINT I; "eine Textzeile"
NEXT
DO: LOOP WHILE INKEY$ = ""    ' Warte, bis eine Taste
                              ' betätigt wird.
CLS 2               ' Lösche nur das Darstellungsfeld.
END
```

***Ausgabe (bevor der Benutzer eine Taste betätigt)***

```
Oberhalb des Text-Darstellungsfeldes, es wird aber nicht gerollt.
 16 eine Textzeile
 17 eine Textzeile
 18 eine Textzeile
 19 eine Textzeile
 20 eine Textzeile

Unterhalb des Text-Darstellungsfeldes.
```

***Ausgabe (nachdem der Benutzter eine Taste betätigt hat)***

```
Oberhalb des Text-Darstellungsfeldes, es wird aber nicht gerollt.




Unterhalb des Text-Darstellungsfeldes.
```

## 3.2 Wie Sie Eingaben von der Tastatur einlesen

Dieser Abschnitt zeigt Ihnen die Verwendung der folgenden Anweisungen und Funktionen, die Ihre BASIC-Programme in die Lage versetzen, Eingaben einzulesen, die per Tastatur eingegeben werden.

- **INPUT**
- **LINE INPUT**
- **INPUT$**
- **INKEY$**

***Hinweis*** Eingaben über die Tastatur werden häufig als "Standardeingabe" bezeichnet. Sie können unter Verwendung des DOS-Symbols < bestimmen, daß die Standardeingabe in Ihr Programm aus einer Datei oder einem anderen Eingabegerät, anstelle der Tastatur, kommen soll. (In Ihrer DOS-Dokumentation finden Sie weitere Informationen zur Datenumleitung.)

## 3.2.1 Die INPUT-Anweisung

Die **INPUT**-Anweisung übernimmt vom Benutzer eingegebene Informationen und speichert diese in einer Liste von Variablen, wie im folgenden Beispiel gezeigt:

```
INPUT A%, B, C$
INPUT D$
PRINT A%, B, C$, D$
```

### *Ausgabe*

```
? 6.6, 45, ein Zeichen
? "zwei, drei"
 7             45           ein Zeichen  zwei, drei
```

Es folgen einige allgemeine Erläuterungen zu **INPUT**:

- **INPUT** selbst fordert den Benutzer zu einer Eingabe mit einem Fragezeichen (?) gefolgt von einem blinkenden Cursor auf.
- Auf die Anweisung **INPUT** folgen ein oder mehrere Variablennamen. Wenn eine Liste von Variablen vorhanden ist, wird jede Variable von einer anderen mit Komma getrennt.
- Die Anzahl von Konstanten, die vom Benutzer nach der **INPUT**-Eingabeaufforderung eingegeben werden, muß die gleiche sein, wie die Anzahl der Variablen in der **INPUT**-Anweisung selbst.
- Die vom Benutzer eingegebenen Werte müssen im Typ mit den Variablen in der auf **INPUT** folgenden Liste übereinstimmen. Mit andern Worten, Sie müssen eine Zahl eingeben, wenn die Variable dazu vorgesehen ist, den Typ Ganzzahl, lange Ganzzahl, einfache Genauigkeit oder doppelte Genauigkeit zu haben. Geben Sie eine Zeichenkette ein, wenn die Variable dazu vorgesehen ist, den Typ Zeichenkette zu haben.
- Da Konstanten in einer Eingabeliste mit Kommata voneinander getrennt sein müssen, sollte eine eingegebene Zeichenkettenkonstante, die eine oder mehrere Kommata enthält, in doppelte Anführungszeichen gesetzt werden. Die doppelten Anführungszeichen stellen sicher, daß die Zeichenkette als Einheit behandelt und nicht in zwei oder mehrere Teile aufgebrochen wird.

Wenn der Benutzer eine der letzten drei Regeln verletzt, gibt BASIC die Fehlermeldung `Korrigieren Sie die Eingabe` aus. Die Meldung wird solange wiederholt, bis die Eingabe sowohl in Anzahl als auch im Typ mit der Variablenliste übereinstimmt.

Wenn Sie Ihre Eingabeaufforderung informativer als mit einem Fragezeichen gestalten wollen, können Sie eine Eingabeaufforderung wie im folgenden Beispiel erstellen:

```
PRINT "Wie lautet die korrekte Uhrzeit ";
INPUT "(Std., Min.)"; Std$, Min$
```

Dies gibt die folgende Aufforderung aus:

```
Wie lautet die korrekte Uhrzeit (Std., Min.)?
```

Beachten Sie das Semikolon zwischen der Eingabeaufforderung und den Eingabevariablen. Dieses Semikolon bewirkt, daß ein Fragezeichen als Teil der Eingabeaufforderung erscheint. Manchmal wollen Sie vielleicht das Fragezeichen unterdrücken; in diesem Falle müssen Sie ein Komma zwischen die Eingabeaufforderung und die Variablenliste schreiben:

```
PRINT "Geben Sie die Uhrzeit ein";
INPUT "(Std., Min.): ", Std$, Min$
```

Dies gibt die folgende Aufforderung aus:

```
Geben Sie die Uhrzeit ein (Std., Min.):
```

## 3.2.2 Die LINE INPUT-Anweisung

Wenn Ihr Programm Textzeilen mit eingeschlossenen Kommata, führenden oder nachfolgenden Leerzeichen akzeptieren soll, und Sie darüberhinaus den Benutzer nicht stets daran erinnern möchten, die Eingabe in doppelte Anführungszeichen zu setzen, verwenden Sie die Anweisung **LINE INPUT**. **LINE INPUT** akzeptiert eine Eingabezeile (abgeschlossen durch Betätigung der EINGABETASTE) von der Tastatur und speichert diese in einer einzigen Zeichenkettenvariablen. Im Unterschied zu **INPUT** gibt die **LINE INPUT**-Anweisung standardmäßig bei einer Eingabeaufforderung kein Fragezeichen aus. Es erlaubt Ihnen jedoch, für die Eingabeaufforderung eine Zeichenkette auszugeben.

Das folgende Beispiel zeigt den Unterschied zwischen **INPUT** und **LINE INPUT**:

```
' Weise die Eingabe drei
' unterschiedlichen Variablen zu:
PRINT "Geben Sie drei durch Kommata getrennte Werte ein";
INPUT ": ",A$,B$,C$
' Weise die Eingabe einer Variablen zu
' (Kommata werden nicht als Trenner zwischen
' Eingaben behandelt):
LINE INPUT "Geben Sie die gleichen drei Werte ein: ",D$
PRINT "A$ = " A$
PRINT "B$ = " B$
PRINT "C$ = " C$
PRINT "D$ = " D$
```

### *Ausgabe*

```
Geben Sie drei durch Kommata getrennte Werte ein: Esel, Katze,
und Hund
Geben Sie die gleichen drei Werte ein: Esel, Katze, und Hund
A$ = Esel
B$ = Katze
C$ = und Hund
D$ = Esel, Katze, und Hund
```

Sowohl mit **INPUT** als auch mit **LINE INPUT** wird eine Eingabe abgeschlossen, wenn der Benutzer die EINGABETASTE betätigt, was den Cursor zusätzlich auf die nächste Zeile bewegt. Wie das nächste Beispiel zeigt, hält ein Semikolon zwischen dem Schlüsselwort **INPUT** und der Eingabeaufforderung den Cursor auf derselben Zeile:

```
INPUT "Erster Wert: ", A
INPUT;"Zweiter Wert: ", B
INPUT "    Dritter Wert: ", C
```

Der folgende Ausschnitt zeigt einige Beispieleingaben für das vorherige Programm und die Positionen der Aufforderungen:

```
Erster Wert: 5
Zweiter Wert: 4    Dritter Wert: 3
```

### 3.2.3 Die INPUT$-Funktion

Sowohl **INPUT** als auch **LINE INPUT** veranlassen Ihr Programm, auf die Betätigung der EINGABETASTE durch den Programmbenutzer zu warten, bevor sie das Eingegebene speichern; das heißt, sie lesen eine Eingabezeile und weisen diese dann den Programmvariablen zu. Im Gegensatz dazu liest die Funktion **INPUT$**(*Zahl*) keine Zeile von der Standardeingabe (normalerweise die Tastatur); stattdessen liest **INPUT$** eine gegebene Anzahl von Zeichen. Die folgende Zeile in einem Programm zum Beispiel liest drei vom Benutzer getippte Zeichen und speichert die Zeichenkette mit drei Zeichen in der Variablen `Test$`:

```
Test$ = INPUT$(3)
```

Im Unterschied zur **INPUT**-Anweisung fordert die **INPUT$**-Funktion den Benutzer weder zur Eingabe von Daten auf, noch gibt sie eingegebene Zeichen auf den Bildschirm aus. Da **INPUT$** darüberhinaus eine Funktion ist, kann sie nicht alleine als komplette Anweisung stehen. **INPUT$** muß in einem Ausdruck erscheinen, wie im folgenden gezeigt:

```
INPUT x             'INPUT ist eine Anweisung.
PRINT INPUT$(1)     'INPUT$ ist eine Funktion, sie muß
Y$ = INPUT$(1)      'daher in einem Ausdruck. erscheinen
```

Die Funktion **INPUT$** liest Eingaben von der Tastatur als eine unformatierte Folge von Zeichen. Anders als **INPUT** oder **LINE INPUT** akzeptiert **INPUT$** jede betätigte Taste einschließlich der Steuerungstasten wie RÜCKTASTE und der ESC-Taste. Beispielsweise weist das fünfmalige Betätigen der EINGABETASTE der Variablen `Test$` in der nächsten Zeile fünf Wagenrücklaufzeichen zu:

```
Test$ = INPUT$(5)
```

### 3.2.4 Die INKEY$-Funktion

Die **INKEY$**-Funktion vervollständigt die Liste der BASIC-Funktionen und -Anweisungen für Tastatureingaben.

Wenn BASIC auf einen Ausdruck trifft, der die **INKEY$**-Funktion enthält, überprüft es, ob der Benutzer nach den folgenden Punkten eine Taste betätigt hat:

- seit dem letzten Finden eines Ausdruckes mit **INKEY$**
- seit dem Programmanfang, beim ersten Auftreten von **INKEY$**

Wenn seit der letzten Überprüfung keine Taste betätigt wurde, gibt **INKEY$** eine leere Zeichenkette ("") zurück. Wenn eine Taste betätigt wurde, gibt **INKEY$** das zu dieser Taste gehörende Zeichen zurück.

### Beispiel

Der wichtigste Unterschied zwischen **INKEY$** und den anderen in diesem Abschnitt erläuterten Anweisungen und Funktionen ist, daß **INKEY$**, während es auf Eingabe überprüft, Ihr Programm mit der Ausführung anderer Dinge fortfahren läßt. Im Gegensatz dazu halten **INPUT**, **LINE INPUT** und **INPUT$** die Programmausführung an, bis eine Eingabe gemacht wird, wie in diesem Beispiel gezeigt:

```
PRINT "Beliebige Taste zum Starten und zum Beenden ";
PRINT "betätigen."
' Tue nichts, bis der Benutzer
' eine Taste betätigt hat:
Beginn$ = INPUT$(1)
I = 1
' Gib die Zahlen von eins bis eine Million aus.
' Überprüfe, ob eine Taste betätigt wird, während
' die Schleife ausgeführt wird:
DO
   PRINT I
   I = I + 1
' Durchlaufe die Schleife, bis der Wert der
' Variablen I größer als eine Million ist, oder bis
' eine Taste betätigt wird:
LOOP UNTIL I > 1000000 OR INKEY$ <> ""
```

## 3.3 Wie Sie den Text-Cursor steuern

Wenn Sie geschriebenen Text auf den Bildschirm ausgeben, markiert der Text-Cursor die Stelle auf dem Bildschirm, an der eine Ausgabe des Programmes – oder vom Benutzer eingetippte Eingaben – als nächstes erscheinen werden. Im nächsten Beispiel wartet der Cursor auf Zeile 1 in Spalte 10 auf Eingabe, nachdem die Anweisung **INPUT** ihre aus neun Zeichen bestehende Eingabeauffforderung, `Vorname:`, ausgegeben hat:

```
' Lösche den Bildschirm und beginne Ausgabe in
' Zeile 1, Spalte 1:
CLS
INPUT "Vorname: ", VorName$
```

Im nächsten Beispiel hält das Semikolon am Ende der zweiten **PRINT**-Anweisung den Cursor auf Zeile 2 in Spalte 32 fest:

```
CLS
PRINT
' Einunddreißig Zeichen sind in der nächsten Zeile:
PRINT "Beliebige Taste zum Fortfahren.";
PRINT INPUT$(1)
```

Die Abschnitte 3.3.1 bis 3.3.3 zeigen, wie die Position des Cursors gesteuert wird, wie seine Form verändert wird, und wie Sie Informationen über seine Position erhalten.

## 3.3.1 Wie Sie den Cursor positionieren

Die bisher erläuterten Anweisungen und Funktionen für Ein- und Ausgabe erlauben keine gute Kontrolle darüber, wo eine Ausgabe angezeigt wird, oder wo der Cursor positioniert wird, nachdem die Ausgabe angezeigt ist. Eingabeanfragen oder Ausgaben beginnen immer in der äußerst linken Spalte des Bildschirms und erstrecken sich jeweils über eine Zeile von oben nach unten, solange nicht ein Semikolon in den **PRINT**- oder **INPUT**-Anweisungen eingesetzt wird, um die Sequenz Wagenrücklauf-Zeilenvorschub zu unterdrücken.

Die in Abschnitt 3.1.4, "Überspringen von Leerzeichen und Springen in eine angegebene Spalte", erläuterten Anweisungen **SPC** und **TAB** ermöglichen einige Kontrolle über die Position des Cursors, indem sie es Ihnen erlauben, den Cursor in jede Spalte innerhalb einer gegebenen Zeile zu bewegen.

Die Anweisung

**LOCATE** [*Zeile*][,[*Spalte*][,[*Cursor*][,[*Beginn*][,*Ende*]]]]

erweitert diese Kontrolle um einen weiteren Schritt, indem sie es Ihnen erlaubt, den Cursor in jeder *Zeile* oder *Spalte* zu positionieren, wie mit der Ausgabe des nächsten Beispiels gezeigt:

```
FOR Zeile = 1 TO 9 STEP 2
   Spalte = 2 * Zeile
   LOCATE Zeile, Spalte
   PRINT "123456789012345678 90";
NEXT
```

***Ausgabe***

```
12345678901234567890
    12345678901234567890
        12345678901234567890
            12345678901234567890
                12345678901234567890
```

## 3.3.2 Wie Sie die Cursor-Größe verändern

Die in der **LOCATE**-Syntax gezeigten optionalen Argumente *Cursor*, *Beginn* und *Ende* erlauben es Ihnen, die Größe des Cursors zu verändern sowie sichtbar oder unsichtbar zu machen. Ein Wert von 1 für *Cursor* macht den Cursor sichtbar, während ein Wert von 0 den Cursor unsichtbar macht. Die Argumente *Beginn* und *Ende* steuern die Höhe des Cursors, wenn dieser eingeschaltet ist, indem sie die oberen und unteren Bildpunktzeilen für den Cursor angeben. (Jedes Zeichen auf dem Bildschirm wird aus Zeilen von Bildpunkten (Pixel), die Lichtpunkte auf dem Bildschirm sind, zusammengesetzt.) Wenn ein Cursor sich über die Höhe einer Textzeile erstreckt, dann hat die Bildpunktzeile des oberen Cursorrandes den Wert 0, während die Bildpunktzeile des unteren Cursorrandes einen Wert von 7 oder 13 hat, abhängig vom Typ Ihres Bildschirms/Adapters (monochrom ist 13, farbig ist 7).

Sie können den Cursor einschalten und seine Form verändern, ohne eine neue Position für ihn anzugeben. Die nächste Anweisung zum Beispiel hält den Cursor dort fest, wo er sich nach Abschluß der nächsten **PRINT**- oder **INPUT**-Anweisung befindet, und macht ihn so hoch wie ein halbes Zeichen:

```
LOCATE ,,1,2,5          ' Die Argumente für Zeile und
                        ' Spalte sind beide optional.
```

Die folgenden Beispiele zeigen unterschiedliche Cursor-Formen, die mit Verwendung unterschiedlicher *Beginn*- und *Ende*-Werte auf einem Farbmonitor erzeugt werden. Jeder in der linken Spalte gezeigten **LOCATE**-Anweisung folgt die Anweisung

```
INPUT "ANFRAGE:", X$
```

| *Anweisung* | *Eingabeaufforderung und Cursor-Form* |
|---|---|
| `LOCATE , , 1, 0, 7` | `Eingabeaufforderung:`  |
| `LOCATE , , 1, 0, 3` | `Eingabeaufforderung:`  |
| `LOCATE , , 1, 5, 7` | `Eingabeaufforderung:`  |
| `LOCATE , , 1, 6, 2` | `Eingabeaufforderung:`  |

Beachten Sie in den vorhergehenden Beispielen, daß die Angabe eines *Beginn*-Arguments, das größer ist als das *Ende*-Argument, einen zweigeteilten Cursor zur Folge hat.

## 3.3.3 Wie Sie Informationen über die Position des Cursors erhalten

Die Funktionen **CSRLIN** und **POS(*n*)** können Sie sich als das Gegenstück der Anweisung **LOCATE** vorstellen; während **LOCATE** dem Cursor angibt, wohin er springen soll, teilen **CSRLIN** und **POS(*n*)** Ihrem Programm mit, wo der Cursor sich befindet. Im einzelnen gibt **CSRLIN** die aktuelle Zeile und **POS(*n*)** die aktuelle Spalte der Cursor-Position an.

Das Argument *n* von **POS(*n*)** wird als "Ersatz"-Argument bezeichnet; das heißt, *n* ist ein Platzhalter, der jeder beliebige numerische Ausdruck sein kann. Zum Beispiel geben sowohl `POS(0)` als auch `POS(1)` den gleichen Wert an.

### Beispiel

Das folgende Beispiel verwendet die Funktion **POS(*n*)**, um 50 Sterne in Zeilen zu je 13 Sternen auszugeben.

```
FOR I% = 1 TO 50
   PRINT "*";     ' Gib einen Stern aus und halte den
                  ' Cursor auf derselben Zeile fest.
   IF POS(1) > 13 THEN PRINT ' Wenn sich die Cursor-
                             ' Position hinter Spalte
                             ' 13 befindet, gehe zur
                             ' nächsten Zeile.
NEXT
```

### *Ausgabe*

```
*************
*************
*************
***********
```

## 3.4 Wie Sie mit Datendateien arbeiten

Datendateien sind physikalische Dateilagen auf Ihrer Diskette/Festplatte, auf denen Informationen permanent gespeichert sind. Die folgenden drei Aufgaben sind bei Verwendung von Datendateien in Ihren BASIC-Programmen beträchtlich vereinfacht:

1. Erstellen, Manipulieren und Speichern großer Mengen von Daten
2. Zugriff auf verschiedene Sätze von Daten mit einem Programm
3. Benutzung desselben Satzes von Daten in mehreren unterschiedlichen Programmen

Die folgenden Abschnitte führen die Konzepte von Sätzen und Feldern ein und stellen unterschiedliche Möglichkeiten vor, mit BASIC auf Datendateien zuzugreifen. Wenn Sie die Abschnitte 3.4.1 bis 3.4.7 abgeschlossen haben, werden Sie die folgenden Punkte bearbeiten können:

- Erstellen neuer Datendateien.
- Öffnen existierender Dateien und Lesen ihrer Inhalte.
- Anfügen neuer Informationen an eine existierende Datendatei.
- Verändern der Inhalte einer existierenden Datendatei.

### 3.4.1 Wie Datendateien organisiert sind

Eine Datendatei ist eine Ansammlung von zusammengehörenden Informationsblöcken, auch "Sätze" genannt. Jeder Satz (Record) in einer Datendatei ist darüber hinaus in "Felder" oder in regelmäßig wiederkehrende Informationsgrößen innerhalb jedes Satzes unterteilt. Wenn Sie eine Datendatei mit einer altmodischeren Möglichkeit zur Informationsspeicherung vergleichen – zum Beispiel einem Aktenordner, der die in einer Firma von Stellenbewerbern ausgefüllten Personalfragebögen enthält – dann entspricht ein Satz einem Fragebogen in diesem Aktenordner. Um den Vergleich einen Schritt weiterzuführen, entspricht ein Feld einer Informationsgröße auf jedem Fragebogen, wie zum Beispiel der Sozialversicherungsnummer.

***Hinweis*** Wenn Sie nicht unter Verwendung von Sätzen auf eine Datei zugreifen wollen, sondern diese stattdessen als eine unformatierte Reihenfolge von Bytes behandeln möchten, dann sollten Sie Abschnitt 3.4.7, "Binärdatei-E/A" lesen.

## 3.4.2 Sequentielle und Direktzugriffsdateien

Die Ausdrücke "sequentielle Datei" und "Direktzugriffsdatei" (Random Access) beziehen sich auf zwei unterschiedliche Möglichkeiten, mit Ihren BASIC-Programmen Daten auf Diskette/Festplatte zu speichern bzw. auf Daten von Diskette/Festplatte zuzugreifen. Eine einfache Möglichkeit, sich diese beiden Arten von Dateien vorzustellen, besteht im folgenden Vergleich: eine sequentielle Datei ist wie ein Tonband, während eine Direktzugriffsdatei gleich einer Langspielplatte ist. Um ein Lied auf einem Tonband zu finden, müssen Sie vom Anfang aus das Band sequentiell vorspulen, bis Sie das Lied, das Sie suchen, gefunden haben – es gibt keine Möglichkeit, das von Ihnen gewünschte Lied direkt anzuspringen. Dies gleicht der Art, mit der Sie Informationen in einer sequentiellen Datei finden: um den 500-sten Satz zu finden, müssen Sie die Sätze 1 bis 499 zuerst lesen.

Wenn Sie im Unterschied dazu ein besonderes Lied auf einer LP hören wollen, müssen Sie nur den Tonarm des Plattenspielers anheben und die Nadel direkt vor dem Lied aufsetzen: Sie können direkt auf alles auf der Platte zugreifen, ohne zunächst alle Lieder vor demjenigen hören zu müssen, das Sie hören möchten. Dies gleicht der Art, mit der Sie jeden Satz, den Sie wünschen, aus einer Direktzugriffsdatei aufrufen können, indem Sie einfach seine Nummer angeben. Die Zugriffszeit wird so erheblich reduziert.

***Hinweis*** Obwohl es keine Möglichkeit gibt, direkt auf einen bestimmten *Satz* in einer sequentiellen Datei zu springen, können Sie mit der Anweisung **SEEK** direkt auf ein bestimmtes *Byte* in der Datei springen (dies entspricht dem "Vorspulen" und "Zurückspulen" des Bandes, um in der vorhergehenden Analogie zu bleiben). Weitere Informationen finden Sie in Abschnitt 3.4.7, "Binärdatei-E/A", und in der Erläuterung der Anweisung **SEEK** im *BASIC-Befehlsverzeichnis*.

## 3.4.3 Wie Sie eine Datendatei öffnen

Bevor Ihr Programm eine Datendatei lesen oder verändern oder ihr Daten hinzufügen kann, muß es zunächst die Datei öffnen. BASIC verwendet hierzu die Anweisung **OPEN**. Die Anweisung **OPEN** erlaubt es Ihrem Programm ebenfalls, eine neue Datei anzulegen.

Die folgende Liste beschreibt die unterschiedlichen Verwendungen der Anweisung **OPEN**:

- Erstellen und öffnen einer neuen Datendatei, so daß ihr Sätze hinzugefügt werden können:

```
' Es befindet sich im aktuellen Verzeichnis keine als
' PREIS.DAT benannte Datei:
OPEN "Preis.Dat" FOR OUTPUT AS #1
```

- Öffnen einer existierenden Datei, so daß neue Sätze bereits vorhandene Daten in der Datei überschreiben:

```
' Eine als PREIS.DAT benannte Datei befindet sich
' bereits in dem aktuellen Verzeichnis; es können neue
' Sätze in sie geschrieben werden, aber alle alten
' Sätze gehen verloren:
OPEN "Preis.Dat" FOR OUTPUT AS #1
```

- Öffnen einer existierenden Datendatei, so daß neue Sätze an das Ende der Datei angefügt werden können, damit die bereits in der Datei befindlichen Daten erhalten bleiben:

```
OPEN "Preis.Dat" FOR APPEND AS #1
```

  Sie können auch den **APPEND**-Modus dazu verwenden, eine neue Datei anzulegen, wenn im aktuellen Verzeichnis noch keine Datei mit dem angegebenen Namen erscheint.

- Öffnen einer existierenden Datendatei, so daß alte Sätze aus ihr gelesen werden können:

```
OPEN "Preis.Dat" FOR INPUT AS #1
```

  Weitere Informationen zu den Modi **INPUT**, **OUTPUT** und **APPEND** finden Sie in Abschnitt 3.4.5, "Wie Sie sequentielle Dateien verwenden".

- Öffnen einer existierenden Datendatei (oder Erstellen einer neuen, wenn eine Datei mit diesem Namen nicht existiert), anschließend Lesen oder Schreiben von Sätzen fester Länge in oder aus der Datei:

```
OPEN "Preis.Dat" FOR RANDOM AS #1
```

  Weitere Informationen über diesen Modus finden Sie in Abschnitt 3.4.6, "Wie Sie Direktzugriffsdateien benutzen".

- Öffnen einer existierenden Datendatei (oder Erstellen einer neuen, wenn eine Datei mit diesem Namen nicht existiert), anschließend Daten aus der Datei lesen oder der Datei neue Daten hinzufügen, beginnend an beliebiger Byte-Position in der Datei:

```
OPEN "Preis.Dat" FOR BINARY AS #1
```

  Weitere Informationen über diesen Modus finden Sie in Abschnitt 3.4.7, "Binärdatei-E/A".

### 3.4.3.1 Dateinummern in BASIC

Die Anweisung **OPEN** umfaßt mehr als nur die Angabe eines Modus für Datei-E/A für eine bestimmte Datei (**OUTPUT, INPUT, APPEND, RANDOM** oder **BINARY**); sie verknüpft ebenso eine einmalige Dateinummer mit dieser Datei. Anschließend wird diese Dateinummer, die jede Ganzzahl von 1 bis 255 sein kann, in dem Programm in nachfolgenden Datei-E/A-Anweisungen als Kurzschreibweise verwendet, um auf die Datei Bezug zu nehmen. Solange die Datei geöffnet ist, bleibt die Nummer mit der Datei verknüpft. Wenn die Datei geschlossen wird, wird die Dateinummer zur Verwendung mit anderen Dateien freigegeben. (In Abschnitt 3.4.4 finden Sie weitere Informationen darüber, wie Dateien geschlossen werden.)

Ihre BASIC-Programme können mehr als eine Datei gleichzeitig öffnen.

Die Funktion **FREEFILE** unterstützt Sie dabei, eine unbenutzte Dateinummer zu finden. **FREEFILE** gibt die nächste verfügbare Nummer an, die in einer **OPEN**-Anweisung mit einer Datei verknüpft werden kann. **FREEFILE** würde zum Beispiel nach den folgenden **OPEN**-Anweisungen den Wert 3 angeben:

```
OPEN "Test1.Dat" FOR RANDOM AS #1
OPEN "Test2.Dat" FOR RANDOM AS #2
DateiNum = FREEFILE
OPEN "Test3.Dat" FOR RANDOM AS #DateiNum
```

Die Funktion **FREEFILE** ist besonders hilfreich, wenn Sie eine eigene Bibliothek schreiben, die Prozeduren enthält, die Dateien öffnen. Mit **FREEFILE** müssen Sie an diese Prozeduren keine Informationen mehr zu der Anzahl offener Dateien übergeben.

### 3.4.3.2 Dateinamen in BASIC

Als Dateiname in **OPEN**-Anweisungen kann jeder Zeichenketten*ausdruck* verwendet werden, der aus jeder Kombination der folgenden Zeichen bestehen kann:

- Die Buchstaben a-z und A-Z
- Die Zahlen 0-9
- Die folgenden Sonderzeichen:
  ( ) { } @ # $ % ^ & ! - _ ' / ~

Der Zeichenkettenausdruck kann auch eine optionale Laufwerksangabe enthalten und auch eine vollständige oder teilweise Pfadangabe. Daher kann Ihr BASIC-Programm mit Datendateien auf anderen Laufwerken oder einem anderen Verzeichnis arbeiten als demjenigem, in dem das Programm läuft. Zum Beispiel sind alle folgenden **OPEN**-Anweisungen gültig.

```
OPEN "..\Rang.Qtr" FOR INPUT As #1
OPEN "a:\Gehalt\1987.man" FOR INPUT AS #2
DateiName$ = "TempDat"
OPEN DateiName$ FOR OUTPUT AS #3
BasisName$ = "Inventur"
OPEN BasisName$ + ".DAT" FOR OUTPUT AS #4
```

DOS besitzt darüber hinaus eigene Einschränkungen für Dateinamen: Sie können nicht mehr als acht Zeichen für den Basisnamen (alles links des optionalen Punktes) und nicht mehr als drei Zeichen für die Erweiterung (alles rechts des optionalen Punktes) verwenden. Lange Dateinamen werden in BASIC auf die folgende Art und Weise gekürzt:

| ***Dateiname im Programm*** | ***Daraus folgender Dateiname in DOS*** |
|---|---|
| `Prog@Dat@Datei` | `PROG@DAT.@DA`<br>Der BASIC-Name ist länger als 11 Zeichen. BASIC nimmt daher die ersten 8 Buchstaben für den Basisnamen, fügt einen Punkt (.) ein und verwendet die nächsten drei Buchstaben als Erweiterung. Alles nach 11 Zeichen wird abgeschnitten. |
| `Post#.Version1` | `POST#.VER`<br>Der Basisname (`Post#`) ist kürzer als acht Zeichen, aber die Erweiterung (`Version1`) ist länger als 3 Zeichen, daher ist die Erweiterung auf drei Zeichen gekürzt. |
| `POST_DATEN.BAK` | Führt zu der Laufzeitfehlermeldung `Unzulässiger Dateiname`. Der Basisname muß kürzer als acht Zeichen sein, wenn Sie ausdrücklich eine Erweiterung verwenden wollen (in diesem Fall `.BAK`). |

DOS arbeitet unabhängig von Groß-/Kleinschreibung, so daß Kleinbuchstaben in Dateinamen alle in Großbuchstaben umgewandelt werden. Deshalb sollten Sie sich nicht auf gemischte Schreibweise von Klein- und Großbuchstaben zur Unterscheidung von Dateien verlassen. Wenn Sie beispielsweise auf der Diskette bereits eine Datei `MESS.DAT` hatten, würde die folgende **OPEN**-Anweisung diese Datei überschreiben und jede bereits in ihr gespeicherte Information zerstören:

```
OPEN "Mess.Dat" FOR OUTPUT AS #1
```

## 3.4.4 Wie Sie Datendateien schließen

Das Schließen einer Datendatei hat zwei wichtige Auswirkungen: erstens schreibt es alle aktuell im Dateipuffer (temporärer Bereich im Speicher) befindlichen Daten in die Datei; zweitens gibt es die mit der Datei verknüpfte Dateinummer zur Benutzung mit einer anderen **OPEN**-Anweisung frei.

Verwenden Sie die Anweisung **CLOSE** innerhalb eines Programmes, um eine Datei zu schließen. Wenn zum Beispiel die Datei `Preis.Dat` mit der folgenden Anweisung geöffnet ist:

```
OPEN "Preis.Dat" FOR OUTPUT AS #1
```

dann beendet die Anweisung `CLOSE #1` die Ausgabe auf `Preis.Dat`. Wenn `Preis.Dat` mit

```
OPEN "Preis.Dat" FOR OUTPUT AS #2
```

geöffnet ist, ist die entsprechende Anweisung zur Beendigung der Ausgabe `CLOSE #2`. Eine **CLOSE**-Anweisung ohne Dateinummernargumente schließt alle offenen Dateien.

Zusätzlich zum Schließen mit einer **CLOSE**-Anweisung kann eine Datendatei ebenso auf Grund der folgenden Umstände geschlossen werden:

- Das BASIC-Programm, das E/A durchführt, wird beendet (Programmbeendigung schließt immer alle offenen Datendateien).
- Das Programm, das E/A durchführt, überträgt mit der Anweisung **RUN** die Kontrolle an ein anderes Programm.

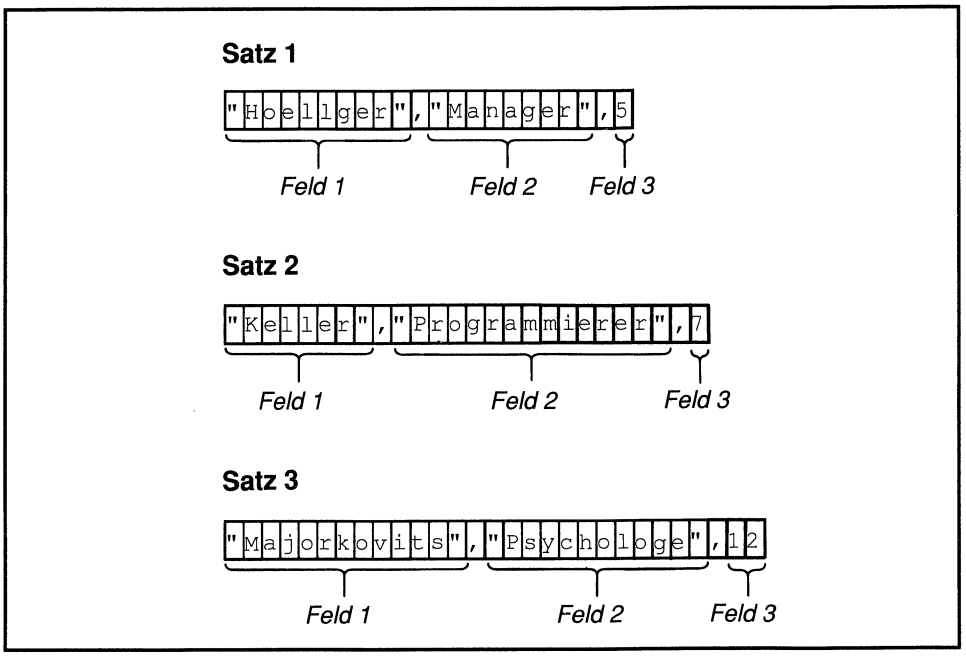
## 3.4.5 Wie Sie sequentielle Dateien verwenden

Dieser Abschnitt beschreibt, wie Sätze in sequentiellen Datendateien organisiert sind, und zeigt Ihnen anschließend, wie Daten aus einer offenen sequentiellen Datei gelesen bzw. wie Daten in eine offene sequentielle Datei geschrieben werden.

### 3.4.5.1 Sätze in sequentiellen Dateien

Sequentielle Dateien sind ASCII(Text)-Dateien. Das heißt, Sie können jedes Textverarbeitungsprogramm zum Betrachten oder Verändern einer sequentiellen Datei verwenden. Sätze werden in sequentiellen Dateien als einzelne Textzeilen gespeichert, die mit einer Wagenrücklauf-Zeilenvorschub (WR-ZV)-Sequenz beendet sind. Jeder Satz ist in Felder geteilt, d.h. er besteht aus Datenstücken, die in jedem Satz in derselben Reihenfolge erscheinen. Abbildung 3.2 zeigt, wie drei Sätze in einer sequentiellen Datei erscheinen könnten.

*Abbildung 3.2 Sätze in sequentiellen Dateien*




Beachten Sie, daß jeder Satz in einer sequentiellen Datei eine andere Länge haben kann; außerdem können Felder in unterschiedlichen Sätzen unterschiedliche Längen haben.

Die Art einer Variablen, in der ein Feld gespeichert ist, bestimmt, wo dieses Feld beginnt und endet. (Beispiele für das Schreiben in Sätze bzw. das Lesen aus Sätzen finden Sie in Abschnitt 3.4.5.2 bis 3.4.5.6.) Wenn Ihr Programm zum Beispiel ein Feld in eine Zeichenkettenvariable einliest, dann kann das Ende dieses Feldes wie folgt angezeigt werden:

- Doppelte Anführungszeichen ("), wenn die Zeichenkette mit doppelten Anführungszeichen beginnt
- Komma (,), wenn die Zeichenkette nicht mit doppelten Anführungszeichen beginnt
- WR-ZV, wenn sich das Feld am Ende des Satzes befindet

Wenn Ihr Programm dagegen ein Feld in eine numerische Variable einliest, dann kann das Ende dieses Feldes wie folgt angezeigt werden:

- Komma
- Ein oder mehrere Leerzeichen
- WR-ZV

### 3.4.5.2 Wie Sie einer neuen sequentiellen Datei Daten hinzufügen

Sie können einer neuen sequentiellen Datei Daten hinzufügen, nachdem Sie diese zunächst mit einer **OPEN** *Dateiname* **FOR OUTPUT**-Anweisung geöffnet haben, um Sätze aufzunehmen. Verwenden Sie die **WRITE #**-Anweisung, um Sätze in eine Datei zu schreiben.

Sie können eine sequentielle Datei für Ausgabe (in eine Datei schreiben) oder für Eingabe (aus einer Datei lesen) öffnen, aber nicht beides zur selben Zeit. Wenn Sie in eine sequentielle Datei geschrieben haben und die von Ihnen gespeicherten Daten wieder einlesen wollen, müssen Sie die Datei zunächst schließen und danach wieder für Eingabe öffnen.

#### Beispiel

Das folgende kurze Programm erstellt eine als `Preis.Dat` benannte sequentielle Datei und fügt der Datei anschließend per Tastatur eingegebene Daten hinzu. Die **OPEN**-Anweisung in diesem Programm erstellt zum einen die Datei und bereitet die Datei zum anderen vor, Sätze aufzunehmen. Danach schreibt **WRITE #** jeden Satz in die Datei. Beachten Sie, daß die in der **WRITE #**-Anweisung verwendete Nummer dieselbe ist, die der Datei `Preis.Dat` in der **OPEN**-Anweisung gegeben ist.

```
' Erstelle eine als Preis.Dat benannte Datei und öffne
' sie, um neue Daten aufzunehmen:
OPEN "Preis.Dat" FOR OUTPUT AS #1
DO
   ' Fahre fort, neue Sätze in Preis.Dat zu schreiben,
   ' bis der Benutzer die <EINGABETASTE> betätigt hat,
   ' ohne einen Firmennamen einzugeben:
   PRINT "Firma (Betätigen Sie die <EINGABETASTE> ";
   INPUT  "zum Verlassen): ", Firma$
   IF Firma$ <> "" THEN
      ' Eingabe der anderen Felder des Satzes:
      INPUT "Art: ",       Art$
      INPUT "Größe: ",     Gross$
```

```
      INPUT "Farbe: ",     Farbe$
      INPUT "Menge: ",     Menge
      ' Schreibe den neuen Satz mit der Anweisung WRITE #
      'in die Datei:
      WRITE #1, Firma$, Art$, Gross$, Farbe$, Menge
   END IF
LOOP UNTIL Firma$ = ""
' Schließe Preis.Dat (Ausgabe in die Datei wird
' damit beendet):
CLOSE #1
END
```

***Warnung*** Wenn Sie im vorhergehenden Beispiel bereits eine mit `Preis.Dat` benannte Datei auf der Diskette hatten, würde der in der **OPEN**-Anweisung angegebene Modus **OUTPUT** die bereits vorhandenen Inhalte von `Preis.Dat` löschen, bevor irgendwelche neuen Daten in sie geschrieben werden. Wenn Sie neue Daten an das Ende einer bereits vorhandenen Datei anfügen wollen, ohne deren alte Inhalte zu löschen, müssen Sie den **APPEND**-Modus von **OPEN** verwenden. Weitere Informationen zum **APPEND**-Modus finden Sie in Abschnitt 3.4.5.4, "Wie Sie Daten an eine existierende sequentielle Datei anfügen".

### 3.4.5.3 Wie Sie Daten aus einer existierenden sequentiellen Datei lesen

Sie können Daten aus einer sequentiellen Datei lesen, nachdem Sie diese zunächst mit der Anweisung **OPEN** *Dateiname* **FOR INPUT** geöffnet haben. Verwenden Sie die Anweisung **INPUT #**, um Sätze aus der Datei Feld für Feld zu lesen. (Weitere Informationen zu anderen Dateieingabe-Anweisungen und -Funktionen, die Sie mit einer sequentiellen Datei verwenden können, finden Sie in Abschnitt 3.4.5.6, "Weitere Möglichkeiten, Daten aus einer sequentiellen Datei zu lesen".)

#### Beispiel

Das folgende Programm öffnet die im vorhergehenden Beispiel erstellte Datendatei `Preis.Dat` und liest Sätze aus der Datei, wobei der gesamte Satz auf dem Bildschirm angezeigt wird, wenn die Menge für den Artikel kleiner als der eingegebene Betrag ist.

Die Anweisung `INPUT #1` liest einen Satz Feld für Feld aus `Preis.Dat`, indem sie die Felder des Satzes den Variablen `Firma$`, `Art$`, `Gross$`, `Farbe$` und `Menge` zuweist. Da dies eine sequentielle Datei ist, werden die Sätze nacheinander vom ersten eingetragenen bis zum letzten eingetragenen gelesen.

Die Funktion **EOF** (Dateiende) überprüft, ob der letzte Satz von **INPUT #** gelesen wurde. Wenn der letzte Satz gelesen wurde, gibt **EOF** den Wert -1 (wahr) an, und die Schleife zum Lesen der Daten wird beendet; wenn nicht der letzte Satz gelesen wurde, gibt **EOF** den Wert 0 (falsch) an, und es wird der nächste Satz aus der Datei gelesen.

```
OPEN "Preis.Dat" FOR INPUT AS #1
PRINT "Anzeige aller Artikel unterhalb welchen ";
INPUT "Niveaus"; NeuBest
DO UNTIL EOF(1)
   INPUT #1, Firma$, Art$, Gross$, Farbe$, Menge
   IF Menge < NeuBest THEN
      PRINT  Firma$, Art$, Gross$, Farbe$, Menge
   END IF
LOOP
CLOSE #1
END
```

### 3.4.5.4 Wie Sie Daten an eine existierende sequentielle Datei anfügen

Wie bereits früher erwähnt, können Sie nicht einfach die sequentielle Datei im Ausgabemodus öffnen, wenn Sie eine sequentielle Datei auf Diskette haben und weitere Daten an deren Ende anfügen möchten. Sie zerstören damit ihre aktuellen Inhalte. Stattdessen müssen Sie den Anfügemodus verwenden.

Wenn Sie an die Datei `Preis.Dat` Daten anfügen möchten, ohne die bereits in der Datei befindlichen Daten zu verlieren, verwenden Sie die folgende Form der **OPEN**-Anweisung:

```
OPEN "Preis.Dat" FOR APPEND AS #1
```

Tatsächlich ist **APPEND** immer eine sichere Alternative zu **OUTPUT**, da auch der Anfügemodus ebenso das Erstellen einer neuen Datei erlaubt. Wenn sich also keine als `Preis.Dat` benannte Datei auf Diskette befindet, wird durch das Öffnen im Anfügemodus eine neue Datei diesen Namens angelegt.

### 3.4.5.5 Weitere Möglichkeiten, Daten in eine sequentielle Datei zu schreiben

Die vorhergehenden Beispiele benutzen alle die Anweisungen **WRITE #**, um Sätze in eine sequentielle Datei zu schreiben. Es gibt jedoch eine weitere Anweisung, die Sie zum Schreiben von Sätzen in sequentielle Dateien verwenden können: **PRINT #**.

Die beste Möglichkeit, den Unterschied zwischen diesen beiden Anweisungen zur Datenspeicherung zu zeigen, besteht in der Untersuchung der Inhalte einer mit beiden Anweisungen erstellten Datei. Der folgende kurze Ausschnitt öffnet eine Datei `Test.Dat` und schreibt den gleichen Satz zweimal in sie hinein, einmal mit **WRITE #** und zum zweiten mit **PRINT #**. Nachdem dieses Programm gelaufen ist, können Sie die Inhalte von `Test.Dat` mit dem DOS-Befehl **TYPE** untersuchen.

```
OPEN "test.dat" FOR OUTPUT AS #1
Nm$ = "Penn, Willi"
Abt$ = "Benutzerschulung"
Niveau = 4
Alter = 25
WRITE #1, Nm$, Abt$, Niveau, Alter
PRINT #1, Nm$, Abt$, Niveau, Alter
CLOSE #1
```

### *Ausgabe*

```
"Penn, Willi","Benutzerschulung",4,25
Penn, Willi   Benutzerschulung          4             25
```

Der mit **WRITE #** gespeicherte Satz hat Kommata, die ausdrücklich jedes Feld des Satzes trennen, genauso wie jeder Zeichenkettenausdruck in Anführungszeichen eingeschlossen ist. Auf der anderen Seite hat **PRINT #** ein Abbild des Satzes in die Datei geschrieben, das genauso aussieht, wie es mit einer einfachen **PRINT**-Anweisung auf dem Bildschirm erscheinen würde. Die Kommata in der **PRINT #**-Anweisung werden als "gehe zum nächsten Ausgabebereich" (ein neuer Ausgabebereich erscheint alle 14 Leerzeichen, beginnend mit dem Anfang einer Zeile) interpretiert, und die Zeichenkettenausdrücke sind nicht in Anführungszeichen eingeschlossen.

An diesem Punkt fragen Sie sich vielleicht, was den Unterschied zwischen diesen Ausgabeanweisungen, abgesehen vom Erscheinungsbild der Daten innerhalb der Datei, ausmacht. Die Antwort liegt in dem, was passiert, wenn Ihr Programm Daten aus der Datei mit einer **INPUT #**-Anweisung zurückliest. Im folgenden Beispiel liest das Programm den mit **WRITE #** gespeicherten Satz und gibt die Werte seiner Felder ohne irgendwelche Probleme aus:

```
OPEN "test.dat" FOR INPUT AS #1
' Lies den ersten Satz ein und zeige die Inhalte jedes
' Feldes an:
INPUT #1, Nm$, Abt$, Niveau, Alter
PRINT Nm$, Abt$, Niveau, Alter
' Lies den zweiten Satz ein und zeige die Inhalte
' jedes Feldes an:
INPUT #1, Nm$, Abt$, Niveau, Alter
PRINT Nm$, Abt$, Niveau, Alter
CLOSE #1
```

### *Ausgabe*

```
Penn, Willi   Benutzerschulung          4        25
```

Wenn das Programm jedoch versucht, den nächsten mit **PRINT #** gespeicherten Satz einzulesen, führt dies zu der Fehlermeldung `Eingabe nach Dateiende`. Da das erste Feld nicht in doppelten Anführungszeichen steht, betrachtet die **INPUT #**-Anweisung das Komma zwischen `Penn` und `Willi` als Feldbegrenzer und weist daher der Variablen `Nm$` nur den Nachnamen `Penn` zu. Den Rest der Zeile

```
Willi       Benutzerschulung       4      25
```

liest **INPUT #** dann in die Variable `Abt$`. Da nun der gesamte Satz gelesen ist, bleibt nichts, was in die Variablen `Niveau` und `Alter` gelesen werden kann, übrig. Das Ergebnis ist die Fehlermeldung `Eingabe nach Dateiende`.

Wenn Sie Sätze speichern, die Zeichenkettenausdrücke enthalten, und Sie wollen diese Sätze später mit der Anweisung **INPUT #** lesen, folgen Sie einer dieser beiden Faustregeln.

1. Benutzen Sie die Anweisung **WRITE #**, um die Sätze zu speichern.
2. Wenn Sie die Anweisung **PRINT #** verwenden wollen, erinnern Sie sich daran, daß **PRINT #** weder Kommata zum Trennen der Felder in den Satz schreibt noch Anführungszeichen um Zeichenketten schreibt. Sie müssen diese Feldtrenner selbst in die **PRINT #**-Anweisung schreiben.

   Zum Beispiel können Sie diese Probleme im vorhergehenden Beispiel durch Verwendung von **PRINT #** mit Anführungszeichen, die jedes Feld umschließen, das Zeichenkettenausdrücke enthält, vermeiden:

```
' 34 ist der ASCII-Wert für das doppelte
' Anführungszeichen:
Q$ = CHR$(34)
' Die nächsten vier Anweisungen schreiben alle den
' Satz in die Datei mit doppelten Anführungszeichen um
' jedes Zeichenkettenfeld:
PRINT #1, Q$ Nm$ Q$ Q$ Abt$ Q$ Niveau Alter
PRINT #1, Q$ Nm$ Q$;Q$ Abt$ Q$;Niveau;Alter
PRINT #1, Q$ Nm$ Q$,Q$ Abt$ Q$,Niveau,Alter
WRITE #1, Nm$, Abt$, Niveau, Alter
```

   Vier Sätze, die mit den vorhergehenden Ausgabeanweisungen erstellt sind:

```
"Penn, Willi""Benutzerschulung" 4  25
"Penn, Willi""Benutzerschulung" 4  25
"Penn, Willi" "Benutzerschulung"          4            25
"Penn, Willi","Benutzerschulung",4,25
```

### 3.4.5.6 Weitere Möglichkeiten, Daten aus einer sequentiellen Datei zu lesen

In den vorhergehenden Abschnitten wird **INPUT #** dazu verwendet, einen Satz (eine Zeile von Daten einer Datei) zu lesen, um den hinter **INPUT #** aufgeführten Variablen unterschiedliche Felder des Satzes zuzuweisen. Dieser Abschnitt untersucht alternative Möglichkeiten, Daten aus sequentiellen Dateien sowohl als Sätze (**LINE INPUT #**) als auch als unformatierte Folgen von Bytes (**INPUT$**) zu lesen.

#### Die Anweisung LINE INPUT #

Mit der Anweisung **LINE INPUT #** kann Ihr Programm eine Textzeile genauso lesen, wie sie in einer Datei erscheint, ohne daß Kommata oder Anführungszeichen als Feldbegrenzer interpretiert werden. Dies ist besonders hilfreich in Programmen, die mit ASCII-Textdateien arbeiten.

**LINE INPUT #** liest eine gesamte Zeile aus einer sequentiellen Datei (bis zu einer Sequenz Wagenrücklauf-Zeilenvorschub) in eine *einzelne* Zeichenkettenvariable.

#### Beispiele

Das folgende kurze Programm liest jede Zeile aus der Datei `Kap1.Txt` und schreibt diese Zeile auf den Bildschirm:

```
' Öffne Kap1.Txt für sequentielle Eingabe:
OPEN "Kap1.Txt" FOR INPUT AS #1
' Lies Zeilen solange sequentiell, bis keine mehr in
' der Datei vorhanden sind:
DO UNTIL EOF(1)
   ' Lies eine Zeile aus der Datei und speichere sie
   ' in der Variablen ZeilPuffer$:
   LINE INPUT #1, ZeilPuffer$
   ' Schreibe die Zeile auf den Bildschirm:
   PRINT ZeilPuffer$
LOOP
```

Das vorhergehende Programm kann leicht zu einem dateikopierenden Hilfsprogramm verändert werden, das jede aus der angegebenen Eingabedatei gelesene Zeile in eine andere Datei und nicht auf den Bildschirm schreibt:

```
' Geben Sie Namen von Eingabe- und Ausgabedateien an:
INPUT "zu kopierende Datei: ", DateiName1$
IF DateiName1$ = "" THEN END
INPUT "Name der neuen Datei: ", DateiName2$
```

```
IF DateiName2$ = "" THEN END
' Öffne erste Datei für sequentielle Eingabe:
OPEN DateiName1$ FOR INPUT AS #1
' Öffne zweite Datei für sequentielle Ausgabe:
OPEN DateiName2$ FOR OUTPUT AS #2
' Lies sequentiell Zeilen
' aus der ersten Datei, bis keine mehr in der Datei
' sind:
DO UNTIL EOF(1)
   ' Lies eine Zeile aus der ersten Datei und
   ' speichere sie in der Variablen ZeilPuffer$:
   LINE INPUT #1, ZeilPuffer$
   ' Schreibe ZeilPuffer$ in die zweite Datei:
   PRINT #2, ZeilPuffer$
LOOP
```

### Die Funktion INPUT$

Eine weitere Möglichkeit Daten aus sequentiellen Dateien (letztlich aus jeder Datei) zu lesen, bietet die Verwendung der Funktion **INPUT$**. Während **INPUT #** und **LINE INPUT #** Zeile für Zeile aus einer Datei lesen, liest **INPUT$** eine angegebene Anzahl von Zeichen aus einer Datei, wie in den folgenden Beispielen gezeigt:

| *Anweisung* | *Funktion* |
|---|---|
| `X$ = INPUT$(100,#1)` | Liest 100 Zeichen aus Datei Nummer 1 und weist sie alle der Zeichenkettenvariablen `X$` zu. |
| `Test$ = INPUT$(1,#2)` | Liest ein Zeichen aus Datei Nummer 2 und weist es der Zeichenkettenvariablen `Test$` zu. |

Die Funktion **INPUT$** ohne eine Dateinummer liest Eingaben immer von der Standardeingabe (normalerweise die Tastatur).

Die **INPUT$**-Funktion führt das aus, was man unter "Binäreingabe" versteht; das heißt, sie liest eine Datei als eine unformatierte Abfolge von Zeichen. Zum Beispiel erkennt sie eine Sequenz Wagenrücklauf-Zeilenvorschub nicht als Kennzeichen für das Ende einer Eingabeoperation. Daher ist **INPUT$** am geeignetsten, wenn Sie wollen, daß Ihr Programm jedes einzelne Zeichen aus einer Datei liest; oder wenn Sie wollen, daß es Binär- oder Nicht-ASCII-Dateien liest.

### Beispiel

Das folgende Programm kopiert die genannte Binärdatei auf den Bildschirm, indem es nur alphanumerische und Interpunktionszeichen im ASCII-Bereich von 32 bis 126, Tabulatoren, Wagenrückläufe und Zeilenvorschübe ausgibt:

```
' 9 ist der ASCII-Wert für horizontales Tab, 10 ist
' der ASCII-Wert für Zeilenvorschub und 13 ist der
' ASCII-Wert für Wagenrücklauf:
CONST ZEILVORSCH = 10, WAGENRUECK = 13, TABZEICH = 9
INPUT "Name der auszugebenden Datei: ", DateiName$
IF DateiName$ = "" THEN END
OPEN DateiName$ FOR INPUT AS #1
DO UNTIL EOF(1)
   Zeichen$ = INPUT$(1, #1)
   ZeichWert = ASC(Zeichen$)
   SELECT CASE ZeichWert
      CASE 32 TO 126
         PRINT Zeichen$;
      CASE TABZEICH, WAGENRUECK
         PRINT Zeichen$;
      CASE ZEILVORSCH
         IF AltZeichWert <> WAGENRUECK THEN
            PRINT Zeichen$
         ENDIF;
      CASE ELSE
         ' Dies ist keines der Zeichen, für die das
         ' Programm vorgesehen ist, also schreibe
         ' nichts.
    END SELECT
    AltZeichWert = ZeichWert
LOOP
```

## 3.4.6 Wie Sie Direktzugriffsdateien benutzen

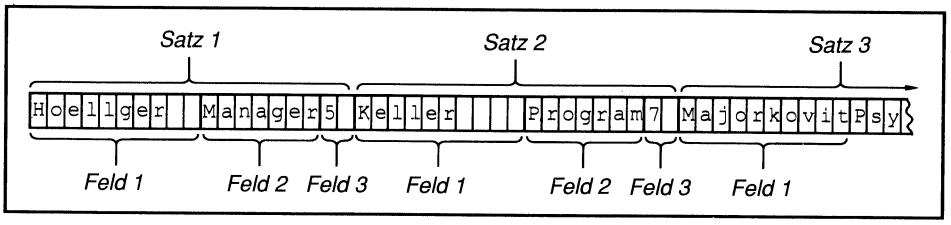
Dieser Abschnitt erläutert, wie Sätze in Direktzugriffsdatendateien organisiert sind, und zeigt Ihnen anschließend, wie Sie Daten aus einer Datei lesen bzw. wie Sie Daten in eine Datei schreiben, die für direkten Zugriff geöffnet ist.

### 3.4.6.1 Sätze in Direktzugriffsdateien

Sätze für Direktzugriff werden anders als sequentielle Sätze gespeichert. Jeder Satz für Direktzugriff ist mit einer festen Länge definiert, genauso wie jedes Feld innerhalb des Satzes. Diese festen Längen bestimmen, wo ein Satz oder Feld beginnt bzw. endet, da zwischen Sätzen keine Felder trennende Kommata oder keine Sequenzen Wagenrücklauf-Zeilenvorschub stehen.

Abbildung 3.3 zeigt, wie drei Sätze in einer Direktzugriffsdatei erscheinen können.

*Abbildung 3.3 Sätze in einer Direktzugriffsdatei*




### Speichern von Zahlen in Sätzen für Direktzugriff

Wenn Sie Sätze speichern, die Zahlen enthalten, sparen Sie mit Direktzugriffsdateien im Vergleich zu sequentiellen Dateien Diskettenplatz. Der Grund dafür ist, daß sequentielle Dateien Zahlen als Abfolge von ASCII-Zeichen, die je eine Ziffer darstellen, speichern, während Direktzugriffsdateien Zahlen in einem komprimierten Binärformat speichern.

Zum Beispiel wird die Zahl 17.000 in einer sequentiellen Datei genauso dargestellt, wie Sie hier auf dieser Seite erscheint. Diese Darstellungsweise verwendet fünf Bytes, eins für jede Ziffer. Wenn 17.000 jedoch als ein ganzzahliges Feld eines Satzes für Direktzugriff gespeichert wird, benötigt es nur zwei Bytes auf der Diskette.

Im allgemeinen benötigen Ganzzahlen (Integer) in Direktzugriffsdateien zwei Bytes, lange Ganzzahlen (Long Integer) und Zahlen einfacher Genauigkeit (Single Precision) benötigen vier Bytes, und Zahlen doppelter Genauigkeit (Double Precision) benötigen acht Bytes.

### 3.4.6.2 Wie Sie einer Direktzugriffsdatei Daten hinzufügen

Ein Programm, das einer Direktzugriffsdatei Daten hinzufügt, erfordert die folgenden drei Schritte:

1. Definieren der Felder für jeden Satz.
2. Öffnen der Datei im Direktzugriffsmodus und Angabe der Länge jedes Satzes.
3. Eingabe für einen neuen Satz einlesen und den Satz in der Datei speichern.

Jeder dieser drei Schritte ist jetzt bedeutend leichter als in BASICA, wie Sie dem folgenden Beispiel entnehmen können.

#### Definieren von Feldern

Sie können Ihren eigenen Satz (Record) mit einer Anweisung **TYPE...END TYPE** definieren, die es Ihnen erlaubt einen zusammengesetzten Datentyp zu erstellen, der Zeichenketten und numerische Elemente mischt. Dies ist ein großer Vorteil gegenüber der früheren Methode, Sätze mit einer **FIELD**-Anweisung einzurichten, die es erforderte, jedes Feld als Zeichenkette zu definieren. Indem Sie einen Satz mit **TYPE...END TYPE** definieren, besteht nicht mehr die Notwendigkeit zur Verwendung der Funktionen, die numerische Daten in Zeichenketten (**MK*Typ*$, MKSMBF$** und **MKDMBF$**) und Zeichenketten in numerische Daten (**CV*Typ*, CVSMBF** und **CVDMBF**) umwandeln.

Der folgende Ausschnitt stellt diese beiden Methoden zur Definition von Sätzen gegenüber:

- Satz definiert mit **TYPE...END TYPE**:

```
' Definiere die Struktur von SatzTyp:
TYPE SatzTyp
   VorName  AS STRING * 10
   NachName AS STRING * 15
   Alter    AS INTEGER
   Gehalt   AS SINGLE
END TYPE
' Deklariere die Variable SatzVar mit dem Typ SatzTyp:
DIM SatzVar AS SatzTyp
.
.
.
```

- Satz mit **FIELD** definiert:

```
' Definiere die Länge der Felder im temporären
' Speicherpuffer:
FIELD #1, 10 AS VorName$,   _
          15 AS NachName$, _
           2 AS Alter$,     _
           4 AS Gehalt$
.
.
.
```

### Öffnen der Datei und Angeben der Satzlänge

Da die Länge eines Satzes für Direktzugriff festgelegt ist, sollten Sie Ihrem Programm mitteilen, wie lang Sie jeden Satz haben wollen; andernfalls ist die Satzlänge standardmäßig 128 Bytes.

Verwenden Sie in der **OPEN**-Anweisung die Klausel **LEN** =, um die Satzlänge anzugeben. Die nächsten beiden Ausschnitte, die mit den oben begonnenen und gegenübergestellten Beispielen fortfahren, zeigen, wie **LEN** = verwendet wird:

- Angeben der Satzlänge für einen Satz, der mit der Anweisung **TYPE...END TYPE** definiert ist:

```
.
.
.
' Öffne die Direktzugriffsdatei und gib die Länge
' eines Satzes als gleich zu der Länge der Variablen
' SatzVar an:
OPEN "ANGEST.DAT" FOR RANDOM AS #1 LEN=LEN(SatzVar)
.
.
.
```

- Angeben der Satzlänge für einen mit **FIELD** definierten Satz:

```
.
.
.
' Öffne die Direktzugriffsdatei und gib die Länge
' eines Satzes an:
OPEN "ANGEST.DAT" FOR RANDOM AS #1 LEN=31
.
.
.
```

Wenn Sie **FIELD** verwenden, müssen Sie, wie Sie sehen, die Längen der Felder selbst addieren (`VorName$` hat 10 Bytes, `NachName$` hat 15 Bytes, `Alter$` hat 2 Bytes, `Gehalt$` hat 4 Bytes, so daß der Satz 10 + 15 + 2 + 4 oder 31 Bytes lang ist). Mit **TYPE...END TYPE** müssen Sie diese Berechnungen nicht länger durchführen. Stattdessen verwenden Sie lediglich die Funktion **LEN**, um die Länge der Variablen, die Sie zur Aufnahme der Sätze festgelegt haben, zu ermitteln (in diesem Fall `SatzVar`).

### Eingeben von Daten in einen Satz und Speichern des Satzes

Sie können Daten direkt in die Elemente eines benutzerdefinierten Satzes eingeben, ohne sich um rechts- und linksbündige Ausrichtung der Eingaben innerhalb eines Feldes mit **LSET** oder **RSET** zu kümmern. Vergleichen Sie die folgenden zwei Ausschnitte, die die oben begonnenen Beispiele fortführen, um zu sehen, wieviel Programmcode Ihnen diese Vorgehensweise erspart:

- Eingeben von Daten in einen Satz für Direktzugriff und Speichern des Satzes mit **TYPE...END TYPE**:

```
.
.
.
' Gib die Daten ein:
INPUT "Vorname";   SatzVar.Vorname
INPUT "Nachname"; SatzVar.NachName
INPUT "Alter";     SatzVar.Alter
INPUT "Gehalt";    SatzVar.Gehalt
' Speichere den Satz:
PUT #1, , SatzVar
.
.
.
```

- Eingeben von Daten in einen Satz für Direktzugriff und Speichern des Satzes mit **FIELD**:

```
.
.
.
' Gib die Daten ein:
INPUT "Vorname";  Vornm$
INPUT "Nachname"; NachNm$
INPUT "Alter";    AlterWert%
INPUT "Gehalt";   GehaltWert!
' Richte die Daten in den Speicherpuffer-Feldern
' linksbündig aus, und wandle Zahlen mit den
' Funktionen MKI$ und MKS$ in Dateizeichenketten um.
LSET VorName$      = Vornm$
LSET NachName$     = NachNm$
LSET Alter$        = MKI$(AlterWert%)
LSET Gehalt$       = MKS$(GehaltWert!)
'Speichere den Satz:
PUT #1
.
.
.
```

## Zusammenfassung

Das folgende Programm faßt alle oben angegebenen Schritte - Definieren von Feldern, Angeben der Satzlänge, Eingeben von Daten und Speichern der eingegebenen Daten - zusammen, um eine als `BESTAND.DAT` benannte Datendatei für Direktzugriff zu öffnen und ihr Sätze anzufügen:

```
DEFINT A-Z
' Definiere die Struktur eines einzelnen Satzes der
' Direktzugriffsdatei. Jeder Satz besteht aus vier
' Feldern fester Länge ("TeilNummer", "Beschreib",
' "EinzPreis", und "Menge"):
TYPE BestPosten
   TeilNummer  AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE
```

```
' Deklariere eine Variable (BestSatz), die den obigen
' Typ verwendet:
DIM BestSatz AS BestPosten
' Öffne die Direktzugriffsdatei, wobei die Länge eines
' Satzes als die Länge der Variablen BestSatz
' festgelegt wird:
OPEN "BESTAND.DAT" FOR RANDOM AS #1 LEN=LEN(BestSatz)
' Verwende LOF(), um die Anzahl der sich bereits in der
' Datei befindenden Sätze zu bestimmen, so daß neue
' Sätze nach diesen angefügt werden:
SatzNummer = LOF(1) \ LEN(BestSatz)
' Füge nun neue Sätze hinzu:
DO
   ' Gib Daten für einen Bestandssatz per Tastatur ein
   ' und speichere die Daten in den unterschiedlichen
   ' Elementen der Variablen BestSatz:
   INPUT "Teilenummer ? ",   BestSatz.TeilNummer
   INPUT "Beschreibung? ",   BestSatz.Beschreib
   INPUT "Einzelpreis ? ",   BestSatz.EinzPreis
   INPUT "Menge       ? ",   BestSatz.Menge
   SatzNummer = SatzNummer + 1
   ' Schreibe die Daten von BestSatz in einen neuen
   ' Satz der Datei:
   PUT #1, SatzNummer, BestSatz
   ' Prüfe, ob weitere Daten gelesen werden sollen:
   INPUT "Weiter (J/N)? ", Antw$
LOOP UNTIL UCASE$(Antw$) = "N"
' Fertig. Schließe die Datei und beende:
CLOSE  #1
END
```

Wenn die Datei BESTAND.DAT bereits existierte, würde dieses Programm weitere Sätze an die Datei anfügen, ohne irgendwelche Sätze, die sich bereits in der Datei befanden, zu überschreiben. Die folgende Schlüsselanweisung macht dies möglich:

```
SatzNummer = LOF(1) \ LEN(BestSatz)
```

Nachfolgend sehen Sie, wie dies funktioniert:

1. Die Funktion `LOF(1)` berechnet die Gesamtzahl von Bytes in der Datei `BESTAND.DAT`. Wenn `BESTAND.DAT` neu ist oder keine Sätze enthält, gibt `LOF(1)` den Wert `0` an.
2. Die Funktion `LEN(BestSatz)` berechnet die Anzahl von Bytes in einem Satz (`BestSatz` ist so definiert, daß sie dieselbe Struktur wie der benutzerdefinierte Typ `BestPosten` hat).
3. Daher sind

$$\text{Anzahl der Sätze} = \frac{\text{Gesamtbytes in Datei}}{\text{Bytes in einem Satz}}$$

Dies ist ein weiterer Vorteil von dem Einsatz eines Satzes fester Länge: Da jeder Satz dieselbe Größe hat, können Sie die obige Formel immer verwenden, um die Anzahl der Sätze in der Datei zu berechnen. Dies funktioniert offensichtlich nicht mit einer sequentiellen Datei, da sequentielle Sätze unterschiedliche Längen haben können.

### 3.4.6.3 Wie Sie Daten sequentiell aus einer Direktzugriffsdatei lesen

Mit der in dem vorhergehenden Abschnitt angegebenen Vorgehensweise zur Berechnung der Anzahl von Sätzen in einer Direktzugriffsdatei können Sie ein Programm schreiben, das alle Sätze in dieser Datei liest.

#### Beispiel

Das folgende Programm liest sequentiell (vom ersten gespeicherten bis zum letzten gespeicherten Satz) Sätze aus der im vorhergehenden Beispiel erstellten Datei `BESTAND.DAT`:

```
' Definiere eine Satzstruktur (Typ) für Sätze einer
' Direktzugriffsdatei:
TYPE BestPosten
   TeilNummer  AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE
' Deklariere eine Variable (BestSatz), die den obigen
' Typ verwendet:
DIM BestSatz AS BestPosten
'Öffne die Direktzugriffsdatei.
OPEN "BESTAND.DAT" FOR RANDOM AS #1 LEN=LEN(BestSatz)
```

```
' Berechne die Anzahl der Sätze in der Datei:
AnzVonSaetzen = LOF(1) \ LEN(BestSatz)
' Lies die Sätze und schreibe die Daten auf den
' Bildschirm:
FOR SatzNummer = 1 TO AnzVonSaetzen
   ' Lies die Daten eines neuen Satzes in die Datei:
   GET #1, SatzNummer, BestSatz
   ' Gib die Daten auf den Bildschirm aus:
   PRINT "Teilenummer  : ", BestSatz.TeilNummer
   PRINT "Beschreibung : ", BestSatz.Beschreib
   PRINT "Einzelpreis  : ", BestSatz.EinzPreis
   PRINT "Menge        : ", BestSatz.Menge
NEXT
' Fertig; schließe die Datei und beende.
CLOSE #1
END
```

***Hinweis*** Es ist nicht notwendig, `BESTAND.DAT` zu schließen, bevor Informationen aus ihr gelesen werden. Das Öffnen einer Datei für Direktzugriff ermöglicht es Ihnen, mit einer einzigen **OPEN**-Anweisung in die Datei zu schreiben oder aus ihr zu lesen.

### 3.4.6.4 Wie Sie Satznummern einsetzen, um auf bestimmte Sätze einer Direktzugriffsdatei zuzugreifen

Sie erhalten jeden Satz einer Direktzugriffsdatei zurück, indem Sie die Nummer des Satzes in einer **GET**-Anweisung aufführen. (Die Satznummer wird einem Satz zugeordnet, wenn dieser mit einer **PUT**-Anweisung gespeichert wird.) Dies ist einer der größten Vorteile, den Direktzugriffsdateien gegenüber sequentiellen Dateien haben, da Ihnen sequentielle Dateien keinen direkten Zugriff auf einen bestimmten Satz ermöglichen.

Die in Abschnitt 3.6.2 aufgeführte Beispielanwendung, **INDEX.BAS**, zeigt eine Methode, einen bestimmten Satz sehr schnell zu finden, indem ein Index der Satznummern durchsucht wird.

### Beispiel

Der folgende Ausschnitt zeigt, wie Sie **GET** mit einer Satznummer verwenden:

```
DEFINT A-Z          ' Standarddatentyp ist Ganzzahl.
CONST FALSCH = 0, WAHR = NOT FALSCH

TYPE BestPosten
   TeilNummer  AS STRING *  6
   Beschreib   AS STRING * 20
   EinzPreis   AS SINGLE
   Menge       AS INTEGER
END TYPE

DIM BestSatz AS BestPosten

OPEN "BESTAND.DAT" FOR RANDOM AS #1 LEN=LEN(BestSatz)

AnzVonSaetzen= LOF(1) \ LEN(BestSatz)
WeitereSaetze = WAHR

DO
   PRINT "Geben Sie die Satznummer des gewünschten ";
   PRINT "Teiles ein: ";
   PRINT "(0 zum Beenden): ";
   INPUT "", SatzNummer

   IF SatzNummer > 0 AND SatzNummer < AnzVonSaetzen    THEN

      ' Lies den Satz, dessen Nummer eingegeben wurde
      ' und speichere ihn in BestSatz:
      GET #1, SatzNummer, BestSatz

      ' Zeige den Satz an:
      .
      .
      .

   ELSEIF SatzNummer = 0 THEN
      WeitereSaetze = FALSCH
   ELSE
      PRINT "Eingegebener Wert außerhalb des Bereichs."
   END IF
LOOP WHILE WeitereSaetze
END
```

## 3.4.7 Binärdatei-E/A

Binärer Zugriff ist die dritte Möglichkeit neben Direktzugriff und sequentiellem Zugriff, um Daten einer Datei zu lesen oder zu schreiben.

Zum Öffnen einer Datei für Binär-E/A verwenden Sie die folgende Anweisung:

**OPEN** *Datei* **FOR BINARY AS** *#Dateinummer*

Binärzugriff ist eine Möglichkeit, an die reinen Bytes einer beliebigen Datei, nicht nur einer ASCII-Datei, heranzukommen. Daher ist Binärzugriff besonders nützlich beim Lesen oder Verändern von Dateien, die in einem Nicht-ASCII-Format gespeichert sind, wie z. B. Microsoft Word-Dateien oder ausführbare (**.EXE**) Dateien. Im Binärmodus geöffnete Dateien werden als eine unformatierte Reihe von Bytes behandelt. Obwohl Sie einen Satz (eine mit einem benutzerdefinierten Typ deklarierte Variable) aus einer im Binärmodus geöffneten Datei lesen bzw. in diese schreiben können, muß die Datei selbst nicht aus Sätzen fester Länge aufgebaut sein. Tatsächlich muß Binär-E/A gar nicht mit Sätzen umgehen, es sei denn, Sie stellen sich jedes Byte in einer Datei als einen separaten Satz vor.

### 3.4.7.1 Vergleich zwischen Binärzugriff und Direktzugriff

Der Modus Binärzugriff gleicht dem Modus Direktzugriff darin, daß Sie nach einer einzigen **OPEN**-Anweisung sowohl aus einer Datei lesen als auch in eine Datei schreiben können. (Binärer Zugriff unterscheidet sich ebenfalls vom sequentiellen Zugriff, bei dem Sie, wenn Sie zwischen Lesen und Schreiben umschalten wollen, eine Datei zuerst schließen und danach neu öffnen müssen, wodurch Sie zum Beginn der Datei zurückkehren.) Außerdem können Sie sich mit Binärzugriff, wie mit Direktzugriff, innerhalb einer geöffneten Datei rückwärts und vorwärts bewegen. Binärzugriff unterstützt sogar dieselben Anweisungen, die zum Lesen und Schreiben von Direktzugriffsdateien verwendet werden:

{**GET** | **PUT**} [#]*Dateinummer*, [*Position*], *Variable*

*Variable* kann hier jeden Typ haben, einschließlich einer Zeichenkette variabler Länge oder eines benutzerdefinierten Typs, und *Position* zeigt auf eine Stelle in der Datei, wo die nächste **GET**- oder **PUT**-Anweisung ausgeführt wird. (Die *Position* ist relativ zum Beginn der Datei; das heißt, das erste Byte in der Datei hat *Position* eins, das zweite Byte hat *Position* zwei usw.) Wenn Sie das Argument *Position* auslassen, bewegen nachfolgende **GET**- bzw. **PUT**-Anweisungen den Dateizeiger sequentiell vom ersten bis zum letzten Byte durch die Datei.

Die **GET**-Anweisung liest aus der Datei eine Anzahl von Bytes, die gleich der Länge von *Variable* ist. Genauso schreibt die **PUT**-Anweisung eine Anzahl von Bytes, die gleich der Länge von *Variable* ist, in die Datei. Wenn *Variable* zum Beispiel den Typ Ganzzahl hat, dann liest **GET** zwei Bytes in *Variable*; wenn sie den Typ einfache Genauigkeit hat, liest **GET** vier Bytes. Wenn Sie kein Argument *Position* in einer **GET**- oder **PUT**-Anweisung angeben, wird der Dateizeiger daher um eine Distanz gleich der Länge von *Variable* weiterbewegt.

***Hinweis*** **GET** und **INPUT$** sind die einzigen Möglichkeiten, Daten aus einer im Binärmodus geöffneten Datei zu lesen.

**PUT** ist die einzige Möglichkeit, Daten in eine im Binärmodus geöffnete Datei zu schreiben.

Anders als Direktzugriff ermöglicht es Ihnen Binärzugriff, sich an jede Byte-Position in einer Datei zu bewegen und anschließend jede von Ihnen gewünschte Anzahl von Bytes zu lesen oder zu schreiben. Im Gegensatz dazu sind Sie bei Direktzugriff gezwungen, sich jedesmal zu einer Satzgrenze zu bewegen und eine festgelegte Anzahl von Bytes (die Länge eines Satzes) zu lesen.

### 3.4.7.2 Wie Sie den Dateizeiger mit SEEK positionieren

Wenn Sie den Dateizeiger ohne gleichzeitige Ausführung irgendeiner E/A auf eine bestimmte Stelle in der Datei bewegen wollen, verwenden Sie die Anweisung

**SEEK** *Dateinummer, Position*

Nach einer **SEEK**-Anweisung beginnt die nächste Lese- oder Schreiboperation in der mit *Dateinummer* geöffneten Datei mit dem in *Position* festgehaltenen Byte.

Das Gegenstück zur **SEEK**-Anweisung ist die Funktion

**SEEK** (*Dateinummer*)

die Ihnen die Byte-Position mitteilt, wo die nächste Lese- oder Schreiboperation beginnt. (Wenn Sie mit Binär-E/A auf eine Datei zugreifen, ergeben die Funktionen **LOC** und **SEEK** ähnliche Ergebnisse, mit dem Unterschied, daß **LOC** die Position des letzten gelesenen oder geschriebenen Bytes angibt, während **SEEK** die Position des nächsten zu lesenden oder zu schreibenden Bytes angibt.)

***Hinweis*** Die **SEEK**-Anweisung und -Funktion arbeiten ebenfalls mit Dateien, die für sequentiellen oder direkten Zugriff geöffnet sind.

Bei sequentiellem Zugriff verhalten sich sowohl die Anweisung als auch die Funktioon genauso, wie sie es bei binärem Zugriff tun; das heißt, die Anweisung **SEEK** bewegt den Dateizeiger auf bestimmte Byte-Positionen, und die Funktion **SEEK** gibt Informationen über das nächste zu lesende oder zu schreibende Byte an.

Wird eine Datei für direkten Zugriff geöffnet, kann die **SEEK**-Anweisung den Dateizeiger jedoch nur auf den Anfang eines Satzes bewegen, nicht aber auf ein Byte innerhalb eines Satzes. Ebenso gibt die Funktion **SEEK** lediglich die Nummer des nächsten Satzes zurück und nicht etwa die Position des nächsten Bytes.

### Beispiel

Das folgende Programm öffnet eine QuickBASIC Quick-Bibliothek; anschließend liest und schreibt es die Namen von BASIC-Prozeduren und anderer externer Symbole, die die Bibliothek enthält. Dieses Programm befindet sich in der Datei **QBIBDRU.BAS** auf Ihren QuickBASIC Original-Disketten.

```
TYPE ExeKopf                    ' Teil des DOS .EXE-Kopfes
                                ' (header)
    andere1 AS STRING * 8       ' Andere Kopfinformation
    ParKopf AS INTEGER          ' Größe des Kopfes in
                                ' Paragraphen
    andere2 AS STRING * 10      ' Andere Kopfinformation
    IP      AS INTEGER          ' Startwert von IP
    CS      AS INTEGER          ' Startwert (relativ) von
END TYPE                        ' CS
TYPE QBKopf                     ' QLB-Kopf
    QBUeber   AS STRING * 6     ' QB-spezifische
                                ' Überschrift
    Magie     AS INTEGER        ' Magisches Wort:
                                ' identifiziert Datei als
                                ' Quick-Bibliothek
    SymbolBeg AS INTEGER        ' Offset vom Kopf zum
                                ' ersten Symbol
END TYPE
```

```
TYPE QbSym                      ' Symboleintrag in Quick-
                                ' Bibliothek
    Flaggen  AS INTEGER         ' Symbolkennzeichen
    NameBeg  AS INTEGER         ' Offset zur Namentabelle
    andere   AS STRING * 4      ' Andere Kopfinformation
END TYPE

DIM EKopf AS ExeKopf, QKopf AS QBKopf, QlbSym AS QbSym,
QKopfPos AS LONG

PRINT "Geben Sie den Namen der Quick-Bibliothek ein";
INPUT ":", DateiName$
DateiName$ = UCASE$(DateiName$)
IF INSTR(DateiName$,".QLB")=0 THEN
    DateiName$=DateiName$ + ".QLB"
ENDIF
OPEN DateiName$ FOR BINARY AS #1

GET #1, , EKopf                 ' Lies den EXE-Formatkopf.

QKopfPos = (EKopf.ParKopf + EKopf.CS) * 16 + EKopf.IP     + 1

GET #1, QKopfPos, QKopf         ' Lies den Quicklib-
                                ' Formatkopf.

IF QKopf.Magie <> &H6C75 THEN
   PRINT "Keine Quick- Bibliothek!" : END
ENDIF

' Berechne die Dateilage des ersten Symboleintrags,
' lies dann diesen Eintrag:
NaechstSym = QKopfPos + QKopf.SymbolBeg
GET #1, NaechstSym, QlbSym
DO
   NaechstSym = SEEK(1)         ' Sichere Dateilage des
                                ' nächsten Symbols.
   AktSym = QKopfPos + QlbSym.NameBeg
   SEEK #1, AktSym              ' Verwende SEEK zur
                                ' Bewegung auf den Namen
                                ' des aktuellen
                                ' Symboleintrags.
   Erwart$ = INPUT$(40, 1)      ' Lies die längste legale
                                ' Zeichenkette plus ein
                                ' zusätzliches Byte für
                                ' das abschließende Null-
                                ' Zeichen (CHR$(0)).
   ' Ziehe den mit Null beendeten Namen heraus:
   SName$ = LEFT$(Erwart$, INSTR(Erwart$, CHR$(0)))
```

```
   ' Gib nur solche Namen aus, die nicht mit "__" oder
   ' "$" beginnen, weil diese Namen normalerweise als
   ' reserviert angesehen werden:
   IF LEFT$(SName$, 2) <> "__" AND LEFT$(SName$, 1) <>
"$" THEN
      PRINT SName$
   END IF
   GET #1, NaechstSym, QlbSym  ' Lies einen
                               ' Symboleintrag.
LOOP WHILE QlbSym.Flaggen      ' Flaggen=0 (falsch)
                               ' bedeutet Ende der
                               ' Tabelle.
```

## 3.5 Wie Sie mit Geräten arbeiten

Microsoft-BASIC unterstützt Geräte-E/A. Das heißt, daß bestimmte Peripherieanschlüsse Ihres Rechners, genauso wie Datendateien auf Diskette, für E/A geöffnet werden können. Eingabe von diesen Geräten oder Ausgabe auf diese Geräte (Device) kann mit den in Tabelle 3.3, "Anweisungen und Funktionen, die für Datendatei-E/A verwendet werden" aufgeführten Anweisungen vorgenommen werden, mit den Ausnahmen, die in Abschnitt 3.5.1, "Unterschiede zwischen Geräte-E/A und Datei-E/A" gezeigt werden.

Die folgende Tabelle führt die von BASIC unterstützten Geräte auf.

| *Name* | *Gerät* | *Unterstützter E/A-Modus* |
|---|---|---|
| **COM1:** | erster serieller Anschluß | Ein- und Ausgabe |
| **COM2:** | zweiter serieller Anschluß | Ein- und Ausgabe |
| **CONS:** | Bildschirm | nur Ausgabe |
| **KYBD:** | Tastatur | nur Eingabe |
| **LPT1:** | erster Drucker | nur Ausgabe |
| **LPT2:** | zweiter Drucker | nur Ausgabe |
| **LPT3:** | dritter Drucker | nur Ausgabe |
| **SCRN:** | Bildschirm | nur Ausgabe |

## 3.5.1 Unterschiede zwischen Geräte- und Datei-E/A

Bestimmte für Datei-E/A verwendete Funktionen und Anweisungen sind für Geräte-E/A nicht erlaubt, während andere Anweisungen und Funktionen sich bei Geräten anders verhalten. Im folgenden werden einige dieser Unterschiede aufgeführt:

- Die Geräte **CONS:**, **SCRN:** und **LPT***n***:** können nicht in den Modi **INPUT** oder **APPEND** geöffnet werden.
- Das Gerät **KYBD:** kann nicht in den Modi **OUTPUT** oder **APPEND** geöffnet werden.
- Die Funktionen **EOF**, **LOC** und **LOF** können nicht mit den Geräten **CONS:**, **KYBD:**, **LPT***n***:** oder **SCRN:** verwendet werden.
- Die Funktionen **EOF**, **LOC** und **LOF** können mit dem seriellen Gerät **COM***n***:** verwendet werden; die Werte, die diese Funktionen angeben, haben jedoch eine andere Bedeutung als die Werte, die sie bei Verwendung mit Datendateien angeben. (In Abschnitt 3.5.2 "Datenübertragung über einen seriellen Anschluß" finden Sie eine Erklärung zur Wirkungweise dieser Funktionen mit **COM***n***:**.)

### Beispiel

Das folgende Programm zeigt, wie die Geräte **LPT1:** oder **SCRN:** mit derselben Syntax, die für Datendateien verwendet wird, für Ausgabe geöffnet werden können. Dieses Programm liest alle Zeilen aus der vom Benutzer gewählten Datei und gibt diese Zeilen anschließend abhängig von der Wahl des Benutzers auf den Bildschirm oder auf den Drucker aus.

```
CLS
' Gib den Namen der zu anzuzeigenden Datei ein:
INPUT "Name der anzuzeigenden Datei: ", DateiNam$
' Wenn kein Name eingegeben wurde, beende das
' Programm, andernfalls öffne die angegebene Datei zum
' Lesen (INPUT):
IF DateiNam$ = "" THEN
    END
 ELSE
    OPEN DateiNam$ FOR INPUT AS #1
ENDIF
' Gib Wahl bei Ausgabegerät der Datei an (Bildschirm oder
' Drucker); wiederhole Eingabeaufforderung solange,
' bis entweder die Tasten "B" oder "D" betätigt werden:
DO
```

```
   ' Gehe zu Zeile 2, Spalte 1 auf dem Bildschirm und
   ' schreibe Anfrage:
   LOCATE 2,1,1
   PRINT "Sende Ausgabe zum Bildschirm (B) "
   PRINT " oder zum    Drucker (D): ";
   ' Überschreibe alles nach der Anfrage:
   PRINT SPACE$(2);
   ' Positioniere den Cursor hinter der Anfrage
   ' und mache ihn sichtbar:
   LOCATE 2,57,1
   Wahl$ = UCASE$(INPUT$(1))      ' Lies Eingabe.
   PRINT Wahl$
LOOP WHILE Wahl$ <> "B" AND Wahl$ <> "D"
' Abhängig von der betätigten Taste den Drucker oder
' den Bildschirm für Ausgabe öffnen:
SELECT CASE Wahl$
   CASE "D"
      OPEN "LPT1:" FOR OUTPUT AS #2
      PRINT "Ausgabe der Datei auf Drucker."
   CASE "B"
      CLS
      OPEN "SCRN:" FOR OUTPUT AS #2
END SELECT
' Setze die Breite des gewählten Ausgabegerätes auf 80
' Spalten:
WIDTH #2, 80
' Solange sich in der Datei Zeilen befinden, lies eine
' Zeile aus der Datei und gib sie auf das gewählte
' Gerät aus:
DO UNTIL EOF(1)
   LINE INPUT #1, ZeilPuffer$
   PRINT #2, ZeilPuffer$
LOOP
CLOSE          ' Beende Eingabe aus der Datei und Ausgabe
               ' auf das Gerät.
END
```

## 3.5.2 Datenübertragung über einen seriellen Anschluß

Die Anweisung **OPEN "COM*n*:"** (wobei *n* 1 oder 2 - falls Sie zwei serielle Anschlüsse haben - sein kann) erlaubt es Ihnen, den/die seriellen Anschluß/Anschlüsse Ihres Computers für serielle (Bit für Bit) Datenübertragung mit anderen Computern oder mit Peripheriegeräten wie Modems oder seriellen Druckern zu öffnen. Die komplette Parameterliste für die Anweisung **OPEN "COM*n*:"** ist in der Erläuterung für **OPEN COM** im *BASIC-Befehlsverzeichnis* beschrieben. Im folgenden werden nur einige Parameter genannt, die Sie angeben können:

- Geschwindigkeit der Datenübertragung, gemessen in "Baud"-Rate (Bits pro Sekunde).
- Ob Übertragungsfehler festgestellt werden sollen, und wie diese Fehler festgestellt werden.
- Wieviele Stopbits (1 oder 2) verwendet werden sollen, um das Ende eines übertragenen Bytes anzuzeigen.
- Wieviele Bits jedes übertragenen oder empfangenen Datenbytes das eigentliche Datum bilden.

Wenn der serielle Anschluß für Datenübertragung mit Peripheriegeräten und anderen Computern geöffnet wird, wird ein Eingabepuffer eingerichtet, der die von dem anderen Gerät gelesenen Bytes aufnimmt. Dies hat seinen Grund darin, daß bei hohen Baud-Raten Zeichen schneller ankommen, als sie verarbeitet werden können. Die Standardgröße für diesen Puffer beträgt 512 Bytes und kann mit der Option **LEN** = *AnzBytes* der Anweisung **OPEN "COM*n*:"** verändert werden. Die von den Funktionen **EOF**, **LOC** und **LOF** angegebenen Werte liefern bei Verwendung mit einem Datenübertragungsgerät Informationen über die Größe dieses Puffers, wie in der folgenden Liste gezeigt:

| ***Funktion*** | ***Angegebene Information*** |
|---|---|
| **EOF** | Ob Zeichen darauf warten, aus dem Eingabepuffer gelesen zu werden. |
| **LOC** | Die Anzahl der im Eingabepuffer wartenden Zeichen. |
| **LOF** | Der Betrag des im Eingabepuffer verbleibenden Platzes (in Bytes). |

Da jedes Zeichen bei einer Datenübertragung wichtig sein kann, haben sowohl **INPUT#** als auch **LINE INPUT #** erhebliche Nachteile beim Lesen von Eingaben anderer Geräte. Dies liegt daran, daß **INPUT #** das Lesen von Daten in eine Variable unterbricht, wenn es auf ein Komma oder eine neue Zeile (und manchmal auf ein Leerzeichen oder ein doppeltes Anführungszeichen) trifft, und **LINE INPUT #** das Lesen von Daten unterbricht, wenn es auf eine neue Zeile trifft. Daher ist **INPUT$** die am besten geeignete Funktion zur Verwendung bei Eingaben von einem Datenübertragungsgerät, da sie alle Zeichen liest.

Die folgende Zeile verwendet die Funktion **LOC**, um den Eingabepuffer auf die Anzahl der dort aus dem als Datei #1 geöffneten Datenübertragungsgerät wartenden Zeichen zu überprüfen. Die Zeile verwendet danach zum Lesen dieser Zeichen die Funktion **INPUT$**, die die Zeichen einer mit `ModemEingabe$` bezeichneten Variablen zuweist:

```
ModemEingabe$ = INPUT$(LOC(1), #1)
```

## 3.6 Anwendungsbeispiele

Die in diesem Abschnitt aufgeführten Anwendungsbeispiele umfassen ein den Bildschirm verwaltendes Programm, das einen Kalender für jeden Monat in jedem Jahr von 1899 bis 2099 ausgibt, ein Datei-E/A-Programm, das einen Index der Satznummern einer Direktzugriffsdatei erstellt und durchsucht, und ein Datenübertragungsprogramm, das Ihren PC dazu veranlaßt, sich nach Anschluß an ein Modem wie eine Datenstation (Terminal) zu verhalten.

### 3.6.1 Immerwährender Kalender (KAL.BAS)

Nach der Aufforderung an den Benutzer, einen Monat von 1 bis 12 und ein Jahr von 1899 bis 2099 einzugeben, gibt das folgende Programm den Kalender für den gegebenen Monat und das angegebene Jahr aus. Die Prozedur `IstSchaltJahr` führt eine geeignete Anpassung des Kalenders für Monate in einem Schaltjahr durch.

#### Verwendete Anweisungen und Funktionen

Dieses Programm demonstriert die folgenden bildschirmverwaltenden Funktionen und Anweisungen, die in Abschnitt 3.1 bis 3.3 erläutert sind:

- **INPUT**
- **INPUT$**
- **LOCATE**
- **POS(0)**
- **PRINT**
- **PRINT USING**
- **TAB**

### Programm-Listing

```
DEFINT A-Z     ' Standardvariablentyp ist Ganzzahl.
' Definiere einen Datentyp für die Namen der Monate
' und die Anzahl von Tagen in jedem Monat:
TYPE MonatTyp
   Anzahl AS INTEGER    ' Anzahl von Tagen in dem Monat
   MName AS STRING * 9 ' Name des Monats
END TYPE
' Deklariere die verwendeten Prozeduren:
DECLARE FUNCTION IstSchaltJahr% (N%)
DECLARE FUNCTION HolEingabe% (Anfrage$, Zeile%,
NiedWert%, HoherWert%)
DECLARE SUB Schreibkalender (Jahr%, Monat%)
DECLARE SUB BerechnMonat (Jahr%, Monat%, StartTag%,
TotalTage%)
DIM MonatDatum(1 TO 12) AS MonatTyp
' Initialisiere Monatsdefinitionen mit den DATA-
' Anweisungen weiter unten.
FOR I = 1 TO 12
   READ MonatDatum(I).MName, MonatDatum(I).Anzahl
NEXT
' Hauptschleife, wiederhole für soviele Monate wie
' gewünscht:
DO
   CLS
   ' Hole Jahr und Monat als Eingabe:
   Jahr = HolEingabe("Jahr (1899 to 2099): ",1,1899,2099)
   Monat = HolEingabe("Monat (1 to 12): ",2,1,12)
   ' Gib den Kalender aus:
   SchreibKalender Jahr, Monat
   ' Ein weiteres Datum?
   LOCATE 13, 1            ' Plaziere Cursor in Zeile
                           ' 13, Spalte 1.
   PRINT "Neues Datum? ";  ' Halte Cursor auf
                           ' derselben Zeile.
   LOCATE , , 1, 0, 13     ' Schalte Cursor ein und
                           ' laß ihn ein Zeichen hoch
                           ' werden.
   Antw$ = INPUT$(1)       ' Warte auf Betätigen einer
                           ' Taste.
   PRINT Antw$             ' Gib betätigte Taste aus.
LOOP WHILE UCASE$(Antw$) = "J"
END
```

```
' Daten für die Monate eines Jahres:
DATA Januar, 31, Februar, 28, Maerz, 31
DATA April, 30, Mai, 31, Juni, 30, Juli, 31, August,    31
DATA September, 30, Oktober, 31, November, 30, Dezember, 31
'
' ================== BERECHNMONAT ====================
'     Berechne den ersten Tag und die Anzahl der Tage
'     eines Monats
' ====================================================
'
SUB BerechnMonat (Jahr, Monat, StartTag, TotalTage) STATIC
   SHARED MonatDatum() AS MonatTyp
   CONST SCHALT = 366 MOD 7
   CONST NORMAL = 365 MOD 7
   ' Berechne Gesamtzahl der Tage (AnzTage) seit
   ' 1.1.1899.
   ' Beginne mit den ganzen Jahren:
   AnzTage = 0
   FOR I = 1899 TO Jahr - 1
      IF IstSchaltJahr(I) THEN       'Wenn Schaltjahr,
        AnzTage = AnzTage + SCHALT   'addiere 366 MOD 7.
      ELSE                           'Wenn normales
        AnzTage = AnzTage + NORMAL   'Jahr, addiere
      END IF                         '365 MOD 7.
   NEXT
   ' Addiere nun die Tage der ganzen Monate:
   FOR I = 1 TO Monat - 1
      AnzTage = AnzTage + MonatDatum(I).Anzahl
   NEXT
   ' Setze die Anzahl von Tagen in dem gewünschten
   ' Monat:
   TotalTage = MonatDatum(Monat).Anzahl
   ' Gleiche aus, wenn gewünschtes Jahr ein Schaltjahr
   ' ist:
   IF IstSchaltJahr(Jahr) THEN
      ' Wenn nach Februar, addiere eins zu den
      ' gesamten Tagen:
      IF Monat > 2 THEN
         AnzTage = AnzTage + 1
      ' Wenn Februar, addiere eins zu den Tagen des
      ' Monats:
      ELSEIF Monat = 2 THEN
         TotalTage = TotalTage + 1
      END IF
   END IF
```

```
   ' Der 1.1.1899 war ein Sonntag, daher ergibt die
   ' Berechnung von "AnzTage MOD 7" den Wochentag
   ' (Sonntag = 0, Montag = 1, Dienstag = 2, und so
   ' weiter) des ersten Tages für den eingegebenen
   ' Monat:
   StartTag = AnzTage MOD 7
END SUB
'
' ================== HOLEINGABE ====================
'     Fordert zur Eingabe auf und testet dann auf
'     gültigen Bereich
' ==================================================
'
FUNCTION HolEingabe (Anfrage$, Zeile, NiedWert,
HoherWert) STATIC
   ' Positioniere Anfrage in der angegebenen Zeile,
   ' schalte den Cursor ein und laß ihn ein Zeichen
   ' hoch werden:
   LOCATE Zeile, 1, 1, 0, 13
   PRINT Anfrage$;
   ' Sichere Spaltenposition:
   Spalte = POS(0)
   ' Gib Wert ein, bis er innerhalb des Bereiches
   ' liegt:
   DO
      LOCATE Zeile, Spalte    ' Positioniere Cursor an
                              ' das Ende der Anfrage.
      PRINT SPACE$(10)        ' Lösche alles bereits
                              ' vorhandene.
      LOCATE Zeile, Spalte    ' Positoniere Cursor
                              ' hinter Anfrage.
      INPUT "", Wert          ' Gib Wert ohne Anfrage
                              ' ein.
   LOOP WHILE (Wert < NiedWert OR Wert > HoherWert)
   ' Gib gültige Eingabe als Wert der Funktion zurück:
   HolEingabe = Wert
END FUNCTION
'
' ================ ISTSCHALTJAHR ====================
' Bestimmt, ob ein Jahr ein Schaltjahr ist oder nicht
' ==================================================
'
```

```
FUNCTION IstSchaltJahr (N)     STATIC
   ' Wenn das Jahr ohne Rest durch vier aber nicht
   ' durch 100 teilbar ist, oder wenn das Jahr ohne
   ' Rest durch 400 teilbar ist, dann ist es ein
   ' Schaltjahr:
   IstSchaltJahr = (N MOD 4 = 0 AND N MOD 100<>0) OR (N MOD
400=0)
END FUNCTION
'
' ================ SCHREIBKALENDER ==================
'   Gibt einen formatierten Kalender für den Monat
'   in dem gegebenen Jahr aus
' ===================================================
'
SUB SchreibKalender (Jahr, Monat)    STATIC
SHARED MonatDatum() AS MonatTyp
   ' Berechne den ersten Tag (So Mo Di ...) und die
   'Anzahl der Tage des Monats:
   BerechnMonat Jahr, Monat, StartTag, TotalTage
   CLS
   Kopf$ = RTRIM$(MonatDatum(Monat).MName)+ "," + STR$(Jahr)
   ' Berechne Stelle zur Zentrierung von Monat und
   ' Jahr:
   LinkRand = (35 - LEN(Kopf$)) \ 2
   ' Gib Kopf aus:
   PRINT TAB(LinkRand); Kopf$
   PRINT
   PRINT "So   Mo   Di   Mi   Do   Fr    Sa"
   PRINT
   ' Berechne erneut und gib Tab zum ersten Tag des
   ' Monats (So Mo Di ...) aus:
   LinkRand = 5 * StartTag + 1
   PRINT TAB(LinkRand);
   ' Gib die Tage des Monats aus:
   FOR I = 1 TO TotalTage
      PRINT USING "##   "; I;
      ' Springe auf nächste Zeile, wenn Cursor sich
      ' hinter Spalte 32 befindet:
      IF POS(0) > 32 THEN PRINT
   NEXT
END SUB
```

***Ausgabe***

```
          Februar, 1988

So   Mo   Di   Mi   Do   Fr   Sa

      1    2    3    4    5    6
 7    8    9   10   11   12   13
14   15   16   17   18   19   20
21   22   23   24   25   26   27
28   29

Neues Datum?
```

## 3.6.2 Wie Sie eine Direktzugriffsdatei indizieren (INDEX.BAS)

Das folgende Programm verwendet eine Indizierungstechnik, um Sätze in einer Direktzugriffsdatei zu speichern und Sätze wiederzufinden. Jedes Element des Datenfeldes `Index` hat zwei Teile: ein Zeichenkettenfeld (`TeilNummer`) und ein Ganzzahlfeld (`SatzNummer`). Dieses Datenfeld ist im Feld `TeilNummer` alphabetisch sortiert, wodurch ein schnelles Suchen nach einer bestimmten Teilenummer in dem Datenfeld mit einem binären Suchalgorithmus ermöglicht wird.

Das Datenfeld `Index` funktioniert ähnlich wie ein Stichwortverzeichnis in einem Buch. Wenn Sie in einem Buch die Seiten finden wollen, die sich mit einem bestimmten Thema befassen, suchen Sie nach einem Eintrag zu diesem Thema im Stichwortverzeichnis. Der Eintrag weist auf einen Eintrag in dem Buch hin. Ähnlich sucht dieses Programm eine Teilenummer in dem alphabetisch sortierten Datenfeld `Index`. Wenn es die Teilenummer gefunden hat, zeigt die entsprechende Satznummer in dem Feld `SatzNummer` auf den Satz, der alle Informationen zu diesem Teil enthält.

### Verwendete Anweisungen und Funktionen

Dieses Programm zeigt die folgenden beim Zugriff auf Direktzugriffsdateien verwendeten Funktionen und Anweisungen:

- **TYPE...END TYPE**
- **OPEN...FOR RANDOM**
- **GET #**
- **PUT #**
- **LOF**

**Programm-Listing**

```
DEFINT A-Z
' Definiere die symbolischen Konstanten, die in dem
' Programm global verwendet werden:
CONST FALSCH = 0, WAHR = NOT FALSCH
' Definiere eine Verbundstruktur für Sätze einer
' Direktzugriffsdatei:
TYPE BestPosten
   TeilNummer AS STRING * 6
   Beschreib  AS STRING * 20
   EinzPreis  AS SINGLE
   Menge      AS INTEGER
END TYPE
' Definiere eine Verbundstruktur für jedes Element des
' Index:
TYPE IndexTyp
   SatzNummer AS INTEGER
   TeilNummer AS STRING * 6
END TYPE
' Deklariere Prozeduren, die aufgerufen werden:
DECLARE FUNCTION Filter$ (Anfrage$)
DECLARE FUNCTION FindeSatz% (TeilNummer$, SatzVar AS
BestPosten)
DECLARE SUB AddSatz (SatzVar AS BestPosten)
DECLARE SUB GibSatz (SatzVar AS BestPosten)
DECLARE SUB SchreibSatz (SatzVar AS BestPosten)
DECLARE SUB SortIndex ()
DECLARE SUB ZeigTeilNummer ()
' Definiere einen Puffer (der den Typ BestPosten
' verwendet) und definiere und dimensioniere das
' Indexdatenfeld:
DIM BestSatz AS BestPosten, Index(1 TO 100) AS IndexTyp
' Öffne die Direktzugriffsdatei:
OPEN "BESTAND.DAT" FOR RANDOM AS #1 LEN = LEN(BestSatz)
' Berechne Anzahl der Sätze in der Datei:
AnzSaetze = LOF(1) \ LEN(BestSatz)
' Wenn Sätze vorhanden sind, lies sie und bilde den
' Index.
```

```
IF AnzSaetze <> 0 THEN
   FOR SatzNummer = 1 TO AnzSaetze
      ' Lies die Daten eines neuen Satzes der Datei:
      GET #1, SatzNummer, BestSatz
      ' Plaziere Teilenummer und Satznummer im Index:
      Index(SatzNummer).SatzNummer = SatzNummer
      Index(SatzNummer).TeilNummer = BestSatz.TeilNummer
   NEXT
   SortIndex       ' Sortiere Index in der Reihenfolge der
                   ' Teilenummern.
END IF
DO                          ' Schleife des Hauptmenüs
   CLS
   PRINT "(F)üge Sätze hinzu."
   PRINT "(Z)eige Sätze an."
   PRINT "(B)eende Programm."
   PRINT
   LOCATE , , 1
   PRINT "Wählen Sie (F, Z, oder B) hier: ";
   ' Warte, bis Benutzer F, Z oder B betätigt:
   DO
      Wahl$ = UCASE$(INPUT$(1))
   LOOP WHILE INSTR("FZB", Wahl$) = 0
   ' Verzweige entsprechend der Wahl:
   SELECT CASE Wahl$
      CASE "F"
         AddSatz BestSatz
      CASE "Z"
         IF AnzSaetze = 0 THEN
            PRINT : PRINT "Noch keine Sätze in der Datei.";
            PRINT " Weiter mit jeder Taste.";
            Pause$ = INPUT$(1)
         ELSE
            GibSatz BestSatz
         END IF
      CASE "B"           ' Beende Programm.
   END SELECT
LOOP UNTIL Wahl$ = "B"
CLOSE #1    ' Alles fertig; schließe Datei und beende.
END
```

```
'
' ===================== ADDSATZ =====================
'  Fügt an die Datei Sätze aus Tastatureingabe an.
' ===================================================
'
SUB AddSatz (SatzVar AS BestPosten) STATIC
   SHARED Index() AS IndexTyp, AnzSaetze
   DO
      CLS
      INPUT "Teilenummer : ", SatzVar.TeilNummer
      INPUT "Beschreibung: ", SatzVar.Beschreib
      ' Aufruf der FUNCTION Filter$, um Preis & Menge
      ' einzugeben:
      SatzVar.EinzPreis = VAL(Filter$("Einzelpreis : "))
      SatzVar.Menge = VAL(Filter$("Menge     : "))
      AnzSaetze = AnzSaetze + 1
      PUT #1, AnzSaetze, SatzVar
      Index(AnzSaetze).SatzNummer = AnzSaetze
      Index(AnzSaetze).TeilNummer = SatzVar.TeilNummer
      PRINT : PRINT "Weiter anfügen? ";
      OK$ = UCASE$(INPUT$(1))
   LOOP WHILE OK$ = "J"
   SortIndex            ' Sortiere Index-Datei erneut.
END SUB
'
' ===================== FILTER ======================
'       Filtert alle nicht-numerischen Zeichen aus
'       einer Zeichenkette und gibt die gefilterte
'       Zeichenkette zurück.
' ===================================================
'
FUNCTION Filter$ (Anfrage$) STATIC
   WertTemp2$ = ""
   PRINT Anfrage$;   ' Gib die übergebene Anfrage aus.
   INPUT "", WertTemp1$   ' Gib eine Zahl als
                          ' Zeichenkette an.
   KettLaenge = LEN(WertTemp1$)  ' Ermittle die Länge
                                 ' der Zeichenkette.
   FOR I% = 1 TO KettLaenge      ' Gehe Zeichen für
                                 ' Zeichen durch die
                                 ' Zeichenkette.
```

```
      Zeich$ = MID$(WertTemp1$, I%, 1)
      ' Ist das Zeichen ein gültiger Teil einer Zahl
      ' (d. h., eine Ziffer oder ein Dezimalpunkt)?
      ' Wenn ja, füge es an das Ende einer neuen
      ' Zeichenkette:
      IF INSTR("0.0123456789", Zeich$) > 0 THEN
         WertTemp2$ = WertTemp2$ + Zeich$
      ' Andernfalls prüfe, ob es ein kleines "l" ist,
      ' da an Schreibmaschinen gewohnte Benutzer eine
      ' 1 vielleicht auf diese Art eingeben:
      ELSEIF Zeich$ = "l" THEN
         WertTemp2$ = WertTemp2$ + "1" 'Ändere das "l"
                                       'in eine "1".
      END IF
   NEXT I%
   Filter$ = WertTemp2$        ' Gib die gefilterte
                               ' Zeichenkette zurück.
END FUNCTION
'
' ================== FINDESATZ ======================
'     Verwendet einen binären Suchalgorithmus, um
'     einen Satz im Index ausfindig zu machen.
' ===================================================
'
FUNCTION FindeSatz% (Teil$, SatzVar AS BestPosten) STATIC
   SHARED Index() AS IndexTyp, AnzSaetze
   ' Setze Ober- und Untergrenze für die Suche:
   ObSatz = AnzSaetze
   UntSatz = 1
   ' Suche, bis oberer Bereich kleiner als unterster
   ' ist:
   DO UNTIL (ObSatz< UntSatz)
      ' Wähle Mittelpunkt:
      Mittpunkt = (ObSatz + UntSatz) \ 2
      ' Überprüfe, ob dies der gewünschte Satz ist
      ' (RTRIM$() entfernt nachfolgende Leerzeichen
      ' aus einer Zeichenkette fester Länge):
      Test$ = RTRIM$(Index(Mittpunkt).TeilNummer)
      ' Wenn dies so ist, verlasse die Schleife:
      IF Test$ = Teil$ THEN
         EXIT DO
```

```
      ' Andernfalls, wenn das, wonach gesucht wird,
      ' größer ist, bewege unteres Ende nach oben:
      ELSEIF Teil$ > Test$ THEN
        UntSatz = Mittpunkt + 1
      ' Andernfalls bewege oberes Ende nach unten:
      ELSE
         ObSatz = Mittpunkt - 1
      END IF
   LOOP
   ' Wenn Teil gefunden wurde, lies Satz aus der Datei
   ' unter Verwendung des Zeigers im Index ein und
   ' setze FindeSatz% auf WAHR:
  IF Test$ = Teil$ THEN
      GET #1, Index(Mittpunkt).SatzNummer, SatzVar
      FindeSatz% = WAHR
   ' Andernfalls, wenn Teil nicht gefunden wurde,
   ' setze FindeSatz% auf FALSCH:
   ELSE
      FindeSatz% = FALSCH
   END IF
END FUNCTION
'
' ====================== GIBSATZ ======================
'    GIBSATZ ruft zunächst ZEIGTEILNUMMER auf, das
'    ein Menü von Teilenummern im oberen Bereich des
'    Bildschirms ausgibt. Danach fordert GIBSATZ den
'    Benutzer auf, eine Teilenummer einzugeben.
'    Abschließend ruft es die Prozeduren FINDESATZ und
'    SCHREIBSATZ auf, um den gegebenen Satz zu finden
'    und auszugeben.
' ====================================================
'
SUB GibSatz (SatzVar AS BestPosten) STATIC
   CLS
   ZeigTeilNummer   ' Rufe die SUB ZeigTeilNummer auf.
   ' Gib die Daten des angegebenen Satzes im
   ' unteren Teil des Bildschirms aus:
   DO
      PRINT "Geben Sie eine der oben aufgelisteten";
      PRINT " Teilenummern an (oder B zum Beenden) und "
      INPUT "betätigen Sie die <EINGABETASTE>: ", Teil$
```

```
      IF UCASE$(Teil$) <> "Q" THEN
         IF FindeSatz(Teil$, SatzVar) THEN
             SchreibSatz SatzVar
         ELSE
             PRINT "Teil nicht gefunden."
         END IF
      END IF
      PRINT STRING$(40, "_")
   LOOP WHILE UCASE$(Teil$) <> "B"
   VIEW PRINT  ' Stelle gesamten Bildschirm als Text-
               ' Darstellungsfeld wieder her.
END SUB
'
' ================== SCHREIBSATZ ====================
'       Gibt einen Satz auf den Bildschirm aus
' ===================================================
'
SUB SchreibSatz (SatzVar AS BestPosten) STATIC
   PRINT "Teilenummer : "; SatzVar.TeilNummer
   PRINT "Beschreibung: "; SatzVar.Beschreib
   PRINT USING "Einzelpreis:$$###.##"; SatzVar.EinzPreis
   PRINT "Menge    :"; SatzVar.Menge
END SUB
'
' ================ ZEIGTEILNUMMER ==================
'  Gibt einen Index aller Teilenummern im oberen
'  Teil des Bildschirms aus.
' ===================================================
'
SUB ZeigTeilNummer STATIC
   SHARED Index() AS IndexTyp, AnzSaetze
   CONST ANZSPALTEN = 8, SPALTENBREIT = 80 \ ANZSPALTEN
   ' Gib im oberen Teil des Bildschirms ein Menü aus,
   ' das alle Teilenummern für Sätze in der Datei
   ' indiziert. Dieses Menü wird in Spalten gleicher
   ' Länge ausgegeben (Ausnahme ist vielleicht die
   ' letzte Spalte, die kürzer als die anderen sein
   ' kann):
   SpalteLang = AnzSaetze
   DO WHILE SpalteLang MOD ANZSPALTEN
      SpalteLang = SpalteLang + 1
   LOOP
```

```
   SpalteLang = SpalteLang \ ANZSPALTEN
   Spalte = 1
   SatzNummer = 1
   DO UNTIL SatzNummer > AnzSaetze
      FOR Zeile = 1 TO SpalteLang
         LOCATE Zeile, Spalte
         PRINT Index(SatzNummer).TeilNummer
         SatzNummer = SatzNummer + 1
         IF SatzNummer > AnzSaetze THEN EXIT FOR
      NEXT Zeile
      Spalte = Spalte + SPALTENBREIT
   LOOP
   LOCATE SpalteLang + 1, 1
   PRINT STRING$(80, "_")    ' Gib Trennzeile aus.
   ' Rolle Information über die Sätze unter dem
   ' Teilenummern-Menü (auf diese Weise werden die
   ' Teilenummern nicht gelöscht):
   VIEW PRINT SpalteLang + 2 TO 24
END SUB
'
' ====================== SORTINDEX ====================
'             Sortiert den Index nach Teilenummern
' =====================================================
'
SUB SortIndex STATIC
   SHARED Index() AS IndexTyp, AnzSaetze
   ' Setze Vergleichsoffset gleich der Hälfte
   ' der Anzahl der Sätze im Index
   Offset = AnzSaetze \ 2
   ' Wiederhole, bis Offset null wird:
   DO WHILE Offset > 0
      Limit = AnzSaetze - Offset
      DO
          ' Nimm an, kein Wechsel an diesem Offset:
          Wechsel = FALSCH
          ' Vergleiche Elemente und wechsele, wenn
          ' eines außerhalb der Reihenfolge:
          FOR I = 1 TO Limit
              IF Index(I).TeilNummer > Index(I +
Offset).TeilNummer THEN
                SWAP Index(I), Index(I + Offset)
                Wechsel = I
             END IF
          NEXT I
          ' Sortiere im nächsten Schritt nur bis dahin,
          ' wo letzter Wechsel vorgenommen wurde:
```

```
            Limit = Wechsel
        LOOP WHILE Wechsel
        ' Keine Wechsel beim letzten Offset, versuche es
        ' mit einem halb so großen:
        Offset = Offset \ 2
    LOOP
END SUB
```

***Ausgabe***

```
max144    max560    pat300    pat550    rw3300    rw5500    sim101    sim204
max321    max909    pat440    pat761    rw3301    rw5501    sim202    sim789

Geben Sie eine der oben aufgelisteten Teilenummern an (oder B zum Beenden)
und betätigen Sie die <EINGABETASTE>: sim204
Teilenummer : sim204
Beschreibung: Tretroller
Einzelpreis:  $80.00
Menge   : 5

Geben Sie eine der oben aufgelisteten Teilenummern an (oder B zum Beenden)
und betätigen Sie die <EINGABETASTE>: pat551
Teil nicht gefunden.

Geben Sie eine der oben aufgelisteten Teilenummern an (oder B zum Beenden)
und betätigen Sie die <EINGABETASTE>: pat550
Teilenummer : pat550
Beschreibung: t
Einzelpreis:   $0.00
Menge   : 0

Geben Sie eine der oben aufgelisteten Teilenummern an (oder B zum Beenden)
und betätigen Sie die <EINGABETASTE>: b
```

## 3.6.3 Terminal-Emulator (TERMINAL.BAS)

Das folgende Programm wandelt Ihren Computer in ein "dummes" Terminal um; das heißt, es läßt Ihren Computer in Zusammenarbeit mit einem Modem nur als ein E/A-Gerät funktionieren. Dieses Programm verwendet die Anweisung **OPEN "COM1:"** und entsprechende Geräte-E/A-Funktionen, um folgende Dinge zu tun:

1. Erlaubt Ihnen, per Tastatur Eingaben zu machen, und die Zeichen an das Modem zu senden.
2. Zeichen, die von dem Modem zurückgegeben werden, auf den Bildschirm zu schreiben.

Beachten Sie, daß eingegebene Zeichen, die auf dem Bildschirm angezeigt werden, zunächst an das Modem gesendet und dann über den geöffneten Datenübertragungskanal an Ihren Computer zurückgegeben werden. Sie können dies überprüfen, wenn Sie ein Modem mit Receive Detect (RD)- und Send Detect (SD)-Lämpchen haben; sie werden beim eintippen in Reihenfolgen aufleuchten.

Um eine Nummer zu wählen, müßten Sie die Befehle Attention Dial Touch-Tone (ATDT, für Tastentelefon) oder Attention Dial Pulse (ATDP, für Wählscheibentelefon) über die Tastatur eingeben (vorausgesetzt, Sie haben ein Hayes-kompatibles Modem).

Alle anderen zu dem Modem gesendeten Befehle müssen ebenfalls über die Tastatur eingegeben werden.

### Verwendete Anweisungen und Funktionen

Dieses Programm zeigt die folgenden zur Datenübertragung mit einem Modem über den seriellen Anschluß Ihres Computers verwendeten Funktionen und Anweisungen:

- **OPEN "COM1:"**
- **EOF**
- **INPUT$**
- **LOC**

```
DEFINT A-Z
DECLARE SUB Filter (InZeichKett$)
COLOR 7, 1  ' Setze Bildschirmfarbe.
CLS
Ende$ = CHR$(0) + CHR$(16) ' Von INKEY$ angegebener
                           ' Wert, wenn Alt+q betätigt
                           ' wird.
' Gib Anfrage in unterster Bildschirmzeile aus und
' schalte den Cursor ein:
LOCATE 24, 1, 1
PRINT STRING$(80, "_");
LOCATE 25, 1
PRINT TAB(30); "Drücke Alt+q zum Beenden";
VIEW PRINT 1 TO 23     ' Ausgaben zwischen den Zeilen 1
                       ' & 23.
' Eröffne Kommunikation (1200 Baud, keine Parität,
' 8-Bit für Daten, 1 Stopbit, 256-Byte
' Eingabepuffer):
OPEN "COM1:1200,N,8,1" FOR RANDOM AS #1 LEN = 256
```

```
DO              ' Hauptschleife der Datenübertragung.
   TastEingab$ = INKEY$  ' Überprüfe Tastatur.
   IF TastEingab$ = Ende$ THEN    ' Verlasse die
                                  ' Schleife, wenn der
                                  ' Benutzer Alt+q
      EXIT DO                     ' betätigt hat.
   ELSEIF TastEingab$ <> "" THEN ' Andernfalls, wenn
                                  ' der Benutzer eine
                                  ' Taste betätigt hat,
                                  ' sende das einge-
      PRINT #1, TastEingab$;      ' tippte Zeichen zum
   END IF                         ' Modem.
   ' Überprüfe das Modem. Wenn Zeichen warten (EOF(1)
   ' ist wahr), hole diese und gib sie auf den
   ' Bildschirm aus:
   IF NOT EOF(1) THEN
     'LOC(1) gibt die Anzahl der wartenden Zeichen an:
      ModemEingab$ = INPUT$(LOC(1), #1)
      Filter ModemEingab$ ' Filtere Zeilenvorschübe
                          ' und Rückschritte aus,
      PRINT ModemEingab$; ' schreibe dann.
   END IF
LOOP
CLOSE             ' Ende der Datenübertragung.
CLS
END
'
' ====================== FILTER ======================
'     Filtert Zeichen einer eingegebenen Zeichenkette
' ====================================================
'
SUB Filter (InZeichKett$) STATIC
   ' Suche nach Rückschrittzeichen (CHR$(8)) und
   ' kodiere diese erneut in CHR$(29) um (Cursor-Taste
   ' nach links):
   DO
      RueckSchr = INSTR(InZeichKett$, CHR$(8))
      IF RueckSchr THEN
          MID$(InZeichKett$, RueckSchr) = CHR$(29)
      END IF
   LOOP WHILE RueckSchr
```

```
    ' Suche nach Zeilenvorschubzeichen (CHR$(10)) und
    ' entferne alle gefundenen:
    DO
       ZeilVor = INSTR(InZeichKett$, CHR$(10))
       IF ZeilVor THEN
          InZeichKett$ = LEFT$(InZeichKett$, ZeilVor - 1)_
                       + MID$(InZeichKett$, ZeilVor + 1)
       END IF
    LOOP WHILE ZeilVor
END SUB
```

## 3.7 Zusammenfassung der Standard-E/A-Anweisungen

Die folgende Tabelle führt die in BASIC für Standard-E/A (typischerweise Eingabe von der Tastatur und Ausgabe auf den Bildschirm) verwendeten Anweisungen und Funktionen auf.

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Ausgabe von Text (ASCII-Zeichen) auf den Bildschirm | **PRINT** | Gibt Text auf den Bildschirm aus |
| | **PRINT USING** | Gibt formatierten Text auf den Bildschirm aus |
| Verändern der Breite einer Ausgabezeile | **WIDTH** | Verändert die Breite des Bildschirms auf 40 Spalten oder 80 Spalten; steuert auf Computern mit einer EGA oder VGA die Anzahl der Zeilen auf dem Bildschirm (25 oder 43) |
| | **WIDTH "SCRN:"** | Weist auf den Bildschirm ausgegebenen Zeilen eine maximale Länge zu, wenn diese vor einer **OPEN "SCRN:"**-Anweisung verwendet wird |
| Lesen von Tastatur-eingabe | **INKEY$** | Liest ein Zeichen von der Tastatur |
| | **INPUT$** | Liest eine bestimmte Anzahl von Zeichen von der Tastatur und speichert diese in einer einzigen Zeichenkettenvariablen |

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| | **INPUT** | Liest Eingabe von der Tastatur und speichert diese in einer Variablenliste |
| | **LINE INPUT** | Liest eine Eingabezeile von der Tastatur und speichert diese in einer einzigen Zeichenketten-variablen |
| Positionieren des Cursors auf dem Bildschirm | **LOCATE** | Bewegt den Cursor zu einer angegebenen Spalte und Zeile |
| | **SPC** | Überspringt Leerzeichen in einer Ausgabezeile |
| | **TAB** | Zeigt Ausgabe in einer angegebenen Spalte an |
| Erhalten von Informa-tionen über die Position des Cursors | **CSRLIN** | Gibt an, in welcher Zeile der Cursor steht |
| | **POS(*n*)** | Gibt an, in welcher Spalte der Cursor steht |
| Erstellen eines Text-Darstellungsfeldes | **VIEW PRINT** | Setzt die oberste und unterste Zeile zur Anzeige von Textausgabe |

## 3.8 Zusammenfassung der Datei-E/A-Anweisungen

Die folgende Tabelle führt die in der BASIC-Datendatei-Programmierung verwendeten Anweisungen und Funktionen auf.

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Erstellen einer neuen Datei oder Zugreifen auf eine existierende Datei | **OPEN** | Öffnet eine Datei zum Lesen oder zum Speichern von Sätzen (E/A) |
| Schließen einer Datei | **CLOSE** | Beendet E/A-Operationen mit einer Datei |
| Speichern von Daten in eine Datei | **PRINT #** | Speichert Variablen als Satzfelder in einer zuvor geöffneten Datei* |
| | **PRINT USING #** | Ähnlich **PRINT #**, mit dem Unterschied, daß **PRINT USING #** die Satzfelder formatiert* |

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| | **WRITE #** | Speichert Variablen als Satzfelder in einer zuvor geöffneten Datei* |
| | **WIDTH** | Legt eine Standardlänge für jeden Satz einer Datei fest* |
| | **PUT** | Speichert die Inhalte einer benutzerdefinierten Variablen in einer zuvor geöffneten Datei** |
| Daten aus einer Datei lesen | **INPUT #** | Liest Felder eines Satzes und weist jedes Feld des Satzes einer Programmvariablen zu* |
| | **INPUT$** | Liest eine Zeichenkette aus einer Datei |
| | **LINE INPUT #** | Liest einen Satz und speichert diesen in einer einzigen Zeichenkettenvariablen* |
| | **GET** | Liest Daten aus einer Datei und weist die Daten den Elementen einer benutzerdefinierten Variablen zu** |
| Verwalten von Dateien auf Diskette/Festplatte | **FILES** | Gibt eine Liste der Dateien eines angegebenen Verzeichnisses aus |
| | **FREEFILE** | Gibt die nächste verfügbare Dateinummer an |
| | **KILL** | Löscht eine Datei von der Diskette/Festplatte |
| | **NAME** | Verändert den Namen einer Datei |
| Erhalten von Informationen über eine Datei | **EOF** | Überprüft, ob alle Daten aus einer Datei gelesen sind |
| | **FILEATTR** | Gibt die Nummer an, die vom Betriebssystem einer geöffneten Datei zugewiesen ist, und eine Nummer, die den Modus anzeigt, in dem die Datei geöffnet wurde (**INPUT, OUTPUT, APPEND, BINARY** oder **RANDOM**) |

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| | **LOC** | Gibt die aktuelle Position innerhalb einer Datei an. Bei binärem Zugriff ist dies die Byte-Position. Bei sequentiellem Zugriff ist dies die durch 128 dividierte Byte-Position. Bei Direktzugriff ist dies die Satznummer des letzten gelesenen oder geschriebenen Satzes. |
| | **LOF** | Gibt die Anzahl der Bytes einer geöffneten Datei an |
| | **SEEK** (Funktion) | Gibt die Stelle an, an der die nächste E/A-Operation stattfinden wird. Bei Direktzugriff ist dies die Nummer des nächsten zu lesenden oder zu schreibenden Satzes. Bei allen anderen Arten von Dateizugriffen ist dies die Byte-Position des nächsten zu lesenden oder zu schreibenden Bytes. |
| Sich in einer Datei bewegen | **SEEK** (Anweisung) | Setzt die Byte-Position der nächsten Lese- oder Schreib-operation in einer geöffneten Datei |

* Zur Verwendung mit sequentiellen Dateien
** Zur Verwendung mit Binär- oder Direktzugriffsdateien

# 4 Zeichenkettenverarbeitung


4.1 Wie Sie Zeichenketten definieren 4.2

4.2 Zeichenketten variabler und fester Länge 4.4
- 4.2.1 Zeichenketten variabler Länge 4.4
- 4.2.2 Zeichenketten fester Länge 4.5

4.3 Wie Sie Zeichenketten zusammensetzen 4.6

4.4 Wie Sie Zeichenketten vergleichen 4.7

4.5 Wie Sie nach Zeichenketten suchen 4.8

4.6 Wie Sie Teile von Zeichenketten zurückbekommen 4.10
- 4.6.1 Wie Sie Zeichen von der linken Seite einer Zeichenkette zurückbekommen 4.10
- 4.6.2 Wie Sie Zeichen von der rechten Seite einer Zeichenkette zurückbekommen 4.12
- 4.6.3 Wie Sie Zeichen von beliebiger Stelle in einer Zeichenkette zurückbekommen 4.12

4.7 Wie Sie Zeichenketten erzeugen 4.13

4.8 Wie Sie Groß- bzw. Kleinschreibung verändern 4.14

4.9 Zeichenketten und Zahlen 4.15

4.10 Wie Sie Zeichenketten verändern 4.16

4.11 Beispielanwendung: Wie Sie eine Zeichenkette in eine Zahl umwandeln (ZEIINZAH.BAS) 4.17

4.12 Zusammenfassung der Anweisungen und Funktionen zur Zeichenkettenverarbeitung 4.18

Dieses Kapitel zeigt Ihnen, wie Sie Reihenfolgen von ASCII-Zeichen, Zeichenketten genannt, manipulieren. Die Manipulation von Zeichenketten ist unverzichtbar, wenn Sie Textdateien verarbeiten, Daten sortieren oder eingegebene Zeichenkettendaten modifizieren.

Wenn Sie dieses Kapitel gelesen haben, werden Sie wissen, wie die folgenden Aufgaben zur Verarbeitung von Zeichenketten durchgeführt werden:

- Deklarieren von Zeichenketten fester Länge.
- Vergleichen von Zeichenketten und das Sortieren in alphabetischer Reihenfolge.
- Suchen nach einer Zeichenkette in einer anderen.
- Teile einer Zeichenkette separieren.
- Abschneiden führender oder nachfolgender Leerzeichen von einer Zeichenkette.
- Erzeugen einer Zeichenkette durch Wiederholung eines einzelnen Zeichens.
- Verändern der Großbuchstaben einer Zeichenkette in Kleinbuchstaben und umgekehrt.
- Umwandeln numerischer Ausdrücke in Zeichenkettendarstellung und umgekehrt.

## 4.1 Wie Sie Zeichenketten definieren

Eine Zeichenkette ist eine zusammenhängende Abfolge von Zeichen. Beispiele für Zeichen sind die Buchstaben des Alphabets (a-z bzw. A-Z), Interpunktionszeichen wie Kommata (,) oder Fragezeichen (?); und andere Symbole aus den Gebieten Mathematik und Wirtschaft, wie Plus- (+) oder Prozentzeichen (%).

In diesem Kapitel kann sich der Ausdruck "Zeichenkette" auf einen der folgenden Punkte beziehen:

- Eine Zeichenkettenkonstante.

  Zeichenkettenkonstanten werden auf eine der folgenden Arten deklariert:

  1. Durch Einschließen einer Abfolge von Zeichen zwischen doppelten Anführungszeichen, wie in der folgenden **PRINT**-Anweisung:

     ```
     PRINT "Ich verarbeite die Datei. Bitte warten."
     ```

     Dies wird als "literale Zeichenkettenkonstante" bezeichnet.

2. Durch die Gleichsetzung eines Namens mit einer literalen Zeichenkette in einer **CONST**-Anweisung, wie im folgenden gezeigt:

```
'Definiere die Zeichenkettenkonstante MELDUNG:
CONST MELDUNG = "Laufwerk nicht verriegelt."
```

Dies wird als "symbolische Zeichenkettenkonstante" bezeichnet.

- Eine Zeichenkettenvariable.

Zeichenkettenvariablen können auf eine der folgenden Arten deklariert werden:

1. Durch Anhängen des Zeichenkettensuffix ($) an den Variablennamen:

```
LINE INPUT "Geben Sie Ihren Namen ein: "; Puffer$
```

2. Durch Verwendung der Anweisung **DEFSTR**:

```
' Alle Variablen beginnend mit dem Buchstaben "P" sind
' Zeichenketten, es sei denn, sie enden mit einem der
' Suffixe für numerischen Typ (%, &, ! oder #):
DEFSTR P
.
.
.
Puffer = "Ihr Name"    ' Puffer ist eine
                       ' Zeichenkettenvariable.
```

3. Durch Deklaration des Zeichenkettennamens in einer **AS STRING**-Anweisung

```
DIM Puffer1 AS STRING     ' Puffer ist eine Zeichenkette
                          ' variabler Länge.
DIM Puffer2 AS STRING*10  ' Puffer2 ist eine 10-Bytes
                          ' lange Zeichenkette fester
                          ' Länge.
```

- Ein Zeichenkettenausdruck.

Ein Zeichenkettenausdruck ist eine Kombination von Zeichenkettenvariablen, -konstanten und/oder -funktionen.

Natürlich sind die Zeichen einer Zeichenkette im Speicher Ihres Computers nicht in einer Form gespeichert, die für die meisten Menschen erkennbar ist. Stattdessen wird jeder Buchstabe in eine Zahl übersetzt, die als ASCII-Code für dieses Zeichen bezeichnet wird. Ein großes "A" ist beispielsweise als dezimal 65 (oder hexadezimal 41H) gespeichert, während ein kleingeschriebenes "a" als dezimal 97 (oder hexadezimal 61H) gespeichert ist. Eine komplette Liste der ASCII-Codes zu den entsprechenden Zeichen finden Sie in Anhang A, "ASCII-Zeichencodes und Tastaturabfragecodes", im *BASIC-Befehlsverzeichnis.*

Sie können auch die BASIC-Funktion **ASC** verwenden, um den ASCII-Code eines Zeichens festzustellen; zum Beispiel gibt `ASC ("A")` den Wert 65 aus. Die Umkehrung der **ASC**-Funktion ist die **CHR$**-Funktion. **CHR$** hat einen ASCII-Code als Argument und gibt das Zeichen dieses Codes aus; zum Beispiel zeigt die folgende Anweisung das Zeichen @ an.

```
PRINT CHR$ (64)
```

## 4.2 Zeichenketten variabler und fester Länge

In früheren BASIC-Versionen hatten Zeichenketten immer variable Längen. BASIC unterstützt jetzt sowohl Zeichenketten mit variabler als auch Zeichenketten mit fester Länge.

### 4.2.1 Zeichenketten variabler Länge

Zeichenketten variabler Länge sind "elastisch"; das heißt, sie verkürzen und verlängern sich, um unterschiedliche Anzahlen von Zeichen zu speichern (von 0 bis zu einem Maximum von 32.767 Zeichen). Die Länge der Variablen `Temp$` im folgenden Beispiel verändert sich abhängig von der Größe dessen, was in `Temp$` gespeichert ist:

```
Temp$ = "1234567"
' Die Funktion LEN gibt die Länge einer Zeichenkette
' (Anzahl von Zeichen, die diese enthält) an:
PRINT LEN(Temp$)
Temp$ = "1234"
PRINT LEN(Temp$)
```

***Ausgabe***

```
 7
 4
```

### 4.2.2 Zeichenketten fester Länge

Zeichenketten fester Länge werden normalerweise als Verbundelemente in einem mit **TYPE...END TYPE** benutzerdefinierten Datentyp verwendet. Dennoch können sie auch selbst in **COMMON-**, **DIM-**, **REDIM-**, **SHARED-** oder **STATIC**-Anweisungen deklariert sein, wie in der folgenden Anweisung:

```
DIM Puffer AS STRING * 10
```

Zeichenketten fester Länge haben immer, wie der Name schon sagt, eine konstante Länge, unabhängig von der Größe dessen, was in ihnen gespeichert ist. Dies wird durch die Ausgabe des folgenden Beispiels gezeigt:

```
DIM NachName AS STRING * 6
DIM VorName AS STRING * 10

NachName = "Lippmann"
VorName  = "Anne"

PRINT "12345678901234567890123456789O"
PRINT VorName; NachName
PRINT LEN(VorName)
PRINT LEN(NachName)
```

#### *Ausgabe*

```
12345678901234567890123456789O
Anne      Lippma
 10
 6
```

Beachten Sie, daß sich die Längen der Zeichenkettenvariablen `VorName` und `NachName` nicht geändert haben, wie mit den von der Funktion **LEN** angegebenen Werten gezeigt wird (und wie auch durch die sechs Leerzeichen nach dem Namen mit vier Buchstaben, `Anne`, gezeigt wird).

Die Ausgabe des obigen Beispiels zeigt, wie Werte, die Variablen fester Länge zugewiesen werden, linksbündig angeordnet werden und, wenn erforderlich, rechts gekürzt werden. Es ist nicht nötig die **LSET**-Funktion zu verwenden, um Werte in Zeichenketten fester Länge linksbündig anzuordnen; dies geschieht automatisch. Wenn Sie eine Zeichenkette innerhalb einer Zeichenkette fester Länge rechtsbündig anordnen wollen, verwenden Sie **RSET**, wie hier gezeigt:

```
DIM NamePuffer AS STRING * 10
RSET NamePuffer = "Anne"
PRINT "1234567890"
PRINT NamePuffer
```

***Ausgabe***

```
1234567890
      Anne
```

## 4.3 Wie Sie Zeichenketten zusammensetzen

Zwei Zeichenketten können mit dem Operator Plus (+) zusammengesetzt werden. Die dem Pluszeichen folgende Zeichenkette wird der dem Pluszeichen vorhergehenden Zeichenkette angefügt, wie im nächsten Beispiel gezeigt:

```
A$ = "erste Zeichenkette"
B$ = "zweite Zeichenkette"
C$ = A$ + B$
PRINT C$
```

***Ausgabe***

```
erste Zeichenkettezweite Zeichenkette
```

Das Zusammensetzen von Zeichenketten auf diese Art wird als "Verketten" (Konkatenation) bezeichnet.

Beachten Sie, daß im vorhergehenden Beispiel die beiden Zeichenketten ohne dazwischenliegendes Leerzeichen zusammengesetzt sind. Wenn Sie ein Leerzeichen haben wollen, können Sie eine der Zeichenketten `A$` oder `B$` wie folgt füllen:

```
B$ = " zweite Zeichenkette"      'Führendes Leerzeichen
                                 'in B$
```

Da Werte in Zeichenketten fester Länge linksbündig angeordnet sind, könnten Sie zusätzliche Leerzeichen finden, wenn Sie die Zeichenketten zusammensetzen, wie in diesem Beispiel:

```
TYPE NameTyp
   Vor  AS STRING * 10
   Nach AS STRING * 12
END TYPE
DIM NameSatz AS NameTyp
```

```
' Die Konstante "Britta" ist in der Variablen
' NameSatz.Vor linksbündig angeordnet, so daß
' 5 nachfolgende Leerzeichen existieren:
NameSatz.Vor  = "Britta"
NameSatz.Nach = "Schmidt"
' Gib eine Zeile mit Zahlen als Bezug aus:
PRINT "12345678901234567890123456789O"
GesamtName$ = NameSatz.Vor + NameSatz.Nach
PRINT GesamtName$
```

***Ausgabe***

```
12345678901234567890123456789O
Britta    Schmidt
```

Die Funktion **LTRIM$** schneidet führende Leerzeichen von einer Zeichenkette ab, und die **RTRIM$**-Funktion schneidet nachfolgende Leerzeichen ab. (Weitere Informationen zu diesen Funktionen finden Sie in Abschnitt 4.6,"Wie Sie Teile von Zeichenketten zurückbekommen".)

## 4.4 Wie Sie Zeichenketten vergleichen

Sie können Zeichenketten mit den folgenden Vergleichsoperatoren vergleichen:

| *Operator* | *Bedeutung* |
|---|---|
| <> | ungleich |
| = | gleich |
| < | kleiner |
| > | größer |
| <= | kleiner gleich |
| >= | größer gleich |

Ein einzelnes Zeichen ist größer als ein anderes, wenn sein ASCII-Wert größer ist. Zum Beispiel ist der ASCII-Wert des Buchstabens "B" größer als der Wert des Buchstabens "A", so daß der Ausdruck `"B" > "A"` wahr ist.

Beim Vergleich zweier Zeichenketten prüft BASIC die ASCII-Werte sich entsprechender Zeichen. Das erste Zeichen, mit dem sich die beiden Zeichenketten unterscheiden, bestimmt die alphabetische Reihenfolge der Zeichenketten. Zum Beispiel sind die Zeichenketten `"Verdichter"` bzw. `"Verdichtung"` bis zum neunten Zeichen identisch (`"e"` bzw. `"u"`). Da der ASCII-Wert von `"e"` kleiner als der ASCII-Wert von `"u"` ist, ist der Ausdruck `"Verdichter" < "Verdichtung"` wahr. Beachten Sie, daß die ASCII-Werte für Buchstaben der alphabetischen Reihenfolge von A bis Z bzw. a bis z folgen, so daß Sie die oben aufgeführten Vergleichsoperatoren dazu verwenden können, Zeichenketten zu alphabetisieren. Überdies werden Großbuchstaben als kleiner als Kleinbuchstaben betrachtet, so daß in einer sortierten Liste, `"ASCII"` vor `"ascii"` stehen würde.

Wenn es keinen Unterschied zwischen sich entsprechenden Zeichen zweier Zeichenketten gibt, und beide die gleiche Länge haben, dann sind die beiden Zeichenketten gleich. Wenn es keinen Unterschied zwischen sich entsprechenden Zeichen zweier Zeichenketten gibt, eine der Zeichenketten aber länger ist, dann ist die längere Zeichenkette größer als die kürzere Zeichenkette. Zum Beispiel sind `"abc" = "abc"` und `"aaaaaaa" > "aaa"` beides wahre Ausdrücke.

***Hinweis*** Bei dem Vergleich von Zeichenketten sind führende und nachfolgende Leerzeichen von Bedeutung. Die Zeichenkette `"  Test"` ist zum Beispiel kleiner als die Zeichenkette `"Test"`, da ein Leerzeichen (`" "`) kleiner als ein `"T"` ist; andererseits ist die Zeichenkette `"Test  "` größer als die Zeichenkette `"Test"`.
Aufgrund dieser Tatsache sollten Sie beim Vergleich von Zeichenketten fester und variabler Länge vorsichtig sein, da Zeichenketten fester Länge nachfolgende Leerzeichen enthalten können.
Die Funktion **LTRIM$** gibt eine Kopie einer Zeichenkette an, deren führende Leerzeichen entfernt sind. Die Funktion **RTRIM$** gibt eine Kopie einer Zeichenkette an, deren nachfolgende Leerzeichen entfernt sind. (Weitere Informationen zu diesen beiden Funktionen finden Sie in Abschnitt 4.6, "Wie Sie Teile von Zeichenketten zurückbekommen".)

## 4.5 Wie Sie nach Zeichenketten suchen

Eine der häufigsten Aufgaben für die Zeichenkettenverarbeitung ist die Suche nach einer Zeichenkette innerhalb einer anderen Zeichenkette. Die Funktion **INSTR** (*Zeichenkette1*, *Zeichenkette2*) gibt an, ob *Zeichenkette2* in *Zeichenkette1* enthalten ist oder nicht, indem sie die Position des ersten Zeichens in *Zeichenkette1* zurückgibt, ab der beide übereinstimmen. Siehe nächstes Beispiel.

```
ZeichKett1$ = "Eine Textzeile, die 39 Zeichen enthält."
ZeichKett2$ = "Zeichen"
PRINT "         1         2         3         4"
PRINT "12345678901234567890123456789012345678901234567890"
PRINT ZeichKett1$
PRINT ZeichKett2$
PRINT INSTR(ZeichKett1$, ZeichKett2$)
```

### *Ausgabe*

```
         1         2         3         4
12345678901234567890123456789012345678901234567890
Eine Textzeile, die 39 Zeichen enthält.
Zeichen
 24
```

Wenn keine Übereinstimmung gefunden wird (das heißt, *Zeichenkette2* ist keine Teilzeichenkette von *Zeichenkette1*), gibt **INSTR** den Wert 0 an.

Eine Abwandlung der oben gezeigten Schreibweise, **INSTR** (*Position, Zeichenkette1, Zeichenkette2*), erlaubt Ihnen die Angabe der Position, ab der die Suche in *Zeichenkette1* beginnen soll. Zur Verdeutlichung, die folgende Abwandlung des vorhergehenden Beispiels verändert die Ausgabe:

```
' Beginne die Suche nach einer Übereinstimmung mit dem
' 30-ten Zeichen von ZeichKett1$:
PRINT INSTR(30, ZeichKett1$, ZeichKett2$)
```

### *Ausgabe*

```
         1         2         3         4
12345678901234567890123456789012345678901234567890
Eine Textzeile, die 39 Zeichen enthält.
Zeichen
 0
```

Mit anderen Worten, die Zeichenkette `"Zeichen"` erscheint nicht nach dem 30-ten Zeichen von `ZeichKett1$`.

Die Abwandlung **INSTR** (*Position, Zeichenkette1, Zeichenkette2*) ermöglicht es, jedes Vorkommen der *Zeichenkette2* in *Zeichenkette1* zu finden, anstatt nur das erste Vorkommen von *Zeichenkette2* in *Zeichenkette1* festzustellen, wie im nächsten Beispiel gezeigt.

```
ZeichKett1$ = "der Gerd, der taucht nicht."
ZeichKett2$ = "der"
PRINT ZeichKett1$
Start       = 1
AnzUeberein = 0
DO
   Ueberein = INSTR(Start,ZeichKett1$,ZeichKett2$)
   IF Ueberein > 0 THEN
      PRINT TAB(Ueberein) ZeichKett2$
      Start       = Ueberein + 1
      AnzUeberein = AnzUeberein + 1
   END IF
LOOP WHILE UEBEREIN
PRINT "Anzahl der Übereinstimmungen =" AnzUeberein
```

***Ausgabe***

```
der Gerd, der taucht nicht.
der
          der
Anzahl der Übereinstimmungen = 2
```

## 4.6 Wie Sie Teile von Zeichenketten zurückbekommen

Abschnitt 4.3. zeigt, wie Sie Zeichenketten mit dem Pluszeichen zusammensetzen. In BASIC sind mehrere Funktionen verwendbar, die Zeichenketten auftrennen bzw. Teile einer Zeichenkette zurückgeben entweder von der linken Seite, von der rechten Seite oder aus der Mitte einer Zielzeichenkette.

### 4.6.1 Wie Sie Zeichen von der linken Seite einer Zeichenkette zurückbekommen

Die Funktionen **LEFT$** und **RTRIM$** geben Zeichen von der linken Seite einer Zeichenkette zurück.

Die Funktion **LEFT$** (*Zeichenkette, Anzahl*) gibt *Anzahl* Zeichen von der linken Seite der *Zeichenkette* aus. Zum Beispiel:

```
Test$ = "Aktenordner"
' Schreibe die 5 äußerst linken Zeichen von Test$:
PRINT LEFT$(Test$,5)
```

***Ausgabe***

```
Akten
```

***Hinweis*** Im vorhergehenden Beispiel verändert **LEFT$** nicht die ursprüngliche Zeichenkette `Test$`; es gibt nur eine andere Zeichenkette an - eine Kopie eines Teiles von `Test$`.
Keine der in diesem Kapitel vorgestellten Funktionen verändert ihre Argumente.

Die Funktion **RTRIM$** gibt den linken Teil einer Zeichenkette, nach Entfernen aller Leerzeichen am Ende der Zeichenkette, an. Vergleichen Sie zum Beispiel die Ausgaben der nächsten beiden **PRINT**-Anweisungen:

```
PRINT "eine linksbündige Zeichenkette     ";
PRINT "nächste"
PRINT RTRIM$ ("eine linksbündige Zeichenkette     ");
PRINT "nächste"
```

***Ausgabe***

```
eine linksbündige Zeichenkette     nächste
eine linksbündige Zeichenkettenächste
```

**RTRIM$** ist hilfreich zum Vergleich von Zeichenketten fester und variabler Länge, wie im nächsten Beispiel:

```
DIM NameSatz AS STRING * 10
NameSatz = "Fred"
NameTest$ = "Fred"
' Verwende RTRIM$, um alle Leerzeichen von der rechten
' Seite der Zeichenkette fester Länge NameSatz zu
' entfernen, vergleiche sie dann mit der Zeichenkette
' variabler Länge NameTest$:
IF RTRIM$(NameSatz) = NameTest$ THEN
   PRINT "Sie sind gleich"
ELSE
   PRINT "Sie sind nicht gleich"
END IF
```

***Ausgabe***

```
Sie sind gleich
```

### 4.6.2 Wie Sie Zeichen von der rechten Seite einer Zeichenkette zurückbekommen

Die Funktionen **RIGHT$** und **LTRIM$** geben Zeichen von der rechten Seite einer Zeichenkette an.

Die Funktion **RIGHT$** (*Zeichenkette, Anzahl*) gibt *Anzahl* Zeichen von der rechten Seite von *Zeichenkette* an. Zum Beispiel:

```
Test$ = "1234567890"
'Schreibe die 5 äußerst rechten Zeichen von Test$:
PRINT RIGHT$ (Test$,5)
```

***Ausgabe***

```
67890
```

Die Funktion **LTRIM$** gibt den rechten Teil einer Zeichenkette, nach Entfernen aller Leerzeichen am Ende der Zeichenkette, an. Vergleichen Sie zum Beispiel die Ausgaben der nächsten zwei **PRINT**-Anweisungen:

```
PRINT "erst";
PRINT "       eine rechtsbündige Zeichenkette"
PRINT "erst";
LTRIM$("       eine rechtsbündige Zeichenkette")
```

***Ausgabe***

```
erst       eine rechtsbündige Zeichenkette
ersteine rechtsbündige Zeichenkette
```

### 4.6.3 Wie Sie Zeichen von beliebiger Stelle in einer Zeichenkette zurückbekommen

Verwenden Sie die Funktion **MID$**, um eine beliebige Anzahl von Zeichen ab einem beliebigen Punkt aus einer Zeichenkette zurückzubekommen. Die Funktion **MID$** (*Zeichenkette, Beginn, Anzahl*) gibt, beginnend mit dem Zeichen auf Position *Beginn*, die *Anzahl* Zeichen aus *Zeichenkette* an. Zum Beispiel:

```
MID$("**über den Wolken**",12,6)
```

beginnt mit dem zwölften Zeichen (W) der Zeichenkette

```
**über den Wolken**
```

und gibt dieses Zeichen plus der nächsten fünf Zeichen an (Wolken).

### Beispiel

Dieses Beispiel zeigt, wie Sie die Funktion **MID$** verwenden, um Zeichen für Zeichen durch eine Textzeile zu gehen:

```
.
.
.
' Ermittle die Anzahl von Zeichen in der
' Textzeichenkette:
Laenge = LEN(ZeichenKette$)
FOR I = 1 TO Laenge
  ' Nimm das erste Zeichen, dann das zweite, das
  ' dritte usw. bis zum Ende der Zeichenkette:
  Zeichen$ = MID$(ZeichenKette$,I,1)
  ' Werte dieses Zeichen aus:
  .
  .
  .
NEXT
```

## 4.7 Wie Sie Zeichenketten erzeugen

Die einfachste Möglichkeit, eine Zeichenkette, die nur aus einem Buchstaben besteht, der beliebig oft wiederholt werden soll, zu erzeugen, ist die Verwendung der eingebauten Funktion **STRING$**. Die Funktion **STRING$** (*Anzahl, Zeichenkette*) erstellt eine Zeichenkette, die *Anzahl* Zeichen lang ist und die jeweils aus dem ersten Zeichen der *Zeichenkette* besteht. Beispielsweise erzeugt die Anweisung

```
Fuellzeichen$ = STRING$(27,"*")
```

eine Zeichenkette von 27 Sternchen. Für Zeichen, die nicht durch Eintippen erzeugt werden können, wie die Zeichen, deren ASCII-Wert größer als 127 ist, verwenden Sie die alternative Form dieser Funktion, **STRING$**(*Anzahl,Code*). Diese Form erzeugt eine Zeichenkette, die *Anzahl* Zeichen lang ist, wobei jedes Zeichen den ASCII-Wert *Code* hat, wie im folgenden Beispiel:

```
' Schreibe eine Zeichenkette von 10 "@" Zeichen
' (64 ist der ASCII-Code für @):
PRINT STRING$(10, 64)
```

***Ausgabe***

```
@@@@@@@@@@
```

Die Funktion **SPACE$**(*Anzahl*) erzeugt eine Zeichenkette, die aus *Anzahl* Leerzeichen besteht.

## 4.8 Wie Sie Groß- bzw. Kleinschreibung verändern

Sie werden es häufig wünschenswert finden, Großbuchstaben in Kleinbuchstaben umzuwandeln und umgekehrt. Beispielsweise wäre dies nützlich bei der Suche nach bestimmten Zeichenketten in einer großen Datei, in denen es nicht auf Groß- bzw. Kleinschreibung ankommt (in anderen Worten, `HILFE`, `hilfe` oder `Hilfe` würden alle als gleich betrachtet). Diese Funktion ist ebenfalls nützlich, wenn Sie nicht sicher sind, ob ein Benutzer Eingabe in Groß- oder Kleinbuchstaben macht.

Die **UCASE$**- und die **LCASE$**-Funktionen führen folgende Umwandlungen mit einer Zeichenkette durch:

- **UCASE$** gibt eine Kopie der an sie übergebenden Zeichenkette aus, wobei alle Kleinbuchstaben in Großbuchstaben umgewandelt werden.
- **LCASE$** gibt eine Kopie der an sie übergebenden Zeichenkette aus, wobei alle Großbuchstaben in Kleinbuchstaben umgewandelt werden.

Zum Beispiel:

```
Muster$ = "*** Der ASCII-Anhang: eine nützliche Code- _
            Tabelle ***"
PRINT Muster$
PRINT UCASE$(Muster$)
PRINT LCASE$(Muster$)
```

***Ausgabe***

```
*** Der ASCII-Anhang: eine nützliche Code-Tabelle ***
*** DER ASCII-ANHANG: EINE NüTZLICHE CODE-TABELLE ***
*** der ascii-anhang: eine nützliche code-tabelle ***
```

Buchstaben, die bereits Großbuchstaben sind, Umlaute, wie auch Zeichen, die keine Buchstaben sind, werden von **UCASE$** nicht verändert; Kleinbuchstaben und Zeichen, die keine Buchstaben sind, sowie die Umlaute und "ß" werden von der **LCASE$**-Funktion ebenfalls nicht verändert.

## 4.9 Zeichenketten und Zahlen

BASIC erlaubt weder, daß eine Zeichenkette einer numerischen Variablen zugewiesen wird, noch daß ein numerischer Ausdruck einer Zeichenkettenvariablen zugewiesen wird. Zum Beispiel führen beide der folgenden Anweisungen zu der Fehlermeldung `Unverträgliche Datentypen`:

```
TempPuffer$ = 45
Zaehler%    = "45"
```

Verwenden Sie stattdessen die Funktion **STR$**, um die Zeichenkettendarstellung einer Zahl anzugeben oder die Funktion **VAL**, um die numerische Darstellung einer Zeichenkette anzugeben.

```
' Die folgenden Anweisungen sind beide gültig:
TempPuffer$ = STR$(45)
Zaehler%    = VAL ("45")
```

Beachten Sie, daß **STR$** das führende Leerzeichen, das BASIC für positive Zahlen ausgibt, einfügt:

```
FOR I = 0 TO 9
   PRINT STR$(I);
NEXT
```

***Ausgabe***

```
 0 1 2 3 4 5 6 7 8 9
```

Wenn Sie dieses Leerzeichen auslassen wollen, können Sie die Funktion **LTRIM$**, wie unten gezeigt, verwenden:

```
FOR I = 0 TO 9
   PRINT LTRIM$(STR$(I));
NEXT
```

***Ausgabe***

```
0123456789
```

Eine andere Möglichkeit zum Formatieren numerischer Ausgaben ist die Verwendung der Anweisung **PRINT USING**. (Weitere Informationen finden Sie in Abschnitt 3.1, "Wie Sie Text auf den Bildschirm ausgeben", in diesem Handbuch und unter der Beschreibung der Anweisung **PRINT USING** im *BASIC-Befehlsverzeichnis*.)

## 4.10 Wie Sie Zeichenketten verändern

Die in jedem der vorhergehenden Abschnitte erwähnten Funktionen lassen ihre Zeichenkettenargumente unverändert. Veränderungen werden nur mit einer Kopie des Argumentes vorgenommen.

Im Unterschied dazu verändert die **MID$**-Anweisung (nicht zu verwechseln mit der in Abschnitt 4.6.3 "Wie Sie Zeichen von beliebiger Stelle in einer Zeichenkette zurückbekommen", behandelten **MID$**-Funktion) ihr Argument durch Einfügen einer anderen Zeichenkette in das Argument. Die eingefügte Zeichenkette ersetzt einen Teil oder die gesamte ursprüngliche Zeichenkette, wie im folgenden Beispiel gezeigt:

```
Temp$ = "In der Sonne."
PRINT Temp$
' Ersetze das "I" durch ein "A":
MID$(Temp$,1) = "A"
' Ersetze "Sonne." durch "Straße":
MID$(Temp$,8) = "Straße"
PRINT Temp$
```

***Ausgabe***

```
In der Sonne.
An der Straße
```

## 4.11 Anwendungsbeispiel: Wie Sie eine Zeichenkette in eine Zahl umwandeln (ZEIINZAH.BAS)

Das folgende Programm liest eine Zahl, die als Zeichenkette eingegeben ist, filtert alle ungültigen numerischen Zeichen (wie zum Beispiel Kommata) aus der Zeichenkette heraus und wandelt die Zeichenkette dann mit der Funktion **VAL** in eine Zahl um.

### Verwendete Anweisungen und Funktionen

Dieses Programm zeigt die folgenden in diesem Kapitel erläuterten Funktionen zur Verarbeitung von Zeichenketten:

- **INSTR**
- **LEN**
- **MID$**
- **VAL**

### Programm-Listing

```
DECLARE FUNCTION Filter$ (Txt$, FilterZeichKett$)
' Gib eine Zeile ein:
LINE INPUT "Geben Sie eine Zahl mit Kommata ein: ", A$
' Suche nur nach gültigen numerischen Zeichen
' (0123456789.-) in der eingegebenen Zeichenkette:
GueltZahl$ = Filter$(A$, "0123456789.-")
' Wandle die Zeichenkette in eine Zahl um:
PRINT "Zahlenwert = " VAL(GueltZahl$)
END
'
' ====================== FILTER ======================
'      Entfernt aus einer Zeichenkette ungewollte
'      Zeichen, indem sie diese mit einer
'      Filterzeichenkette vergleicht, die nur
'      zulässige numerische Zeichen enthält
' ====================================================
'
FUNCTION Filter$ (Txt$, FilterZeichKett$) STATIC
   Temp$ = ""
   TxtLaenge = LEN(Txt$)
```

```
   FOR I = 1 TO TxtLaenge      ' Isoliere jedes Zeichen
      C$ = MID$(Txt$, I, 1)    ' der Zeichenkette.
      ' Wenn sich das Zeichen in der
      ' Filterzeichenkette befindet, sichere es:
      IF INSTR(FilterZeichKett$, C$) <> 0 THEN
        Temp$ = Temp$ + C$
      END IF
   NEXT I
   Filter$ = Temp$
END FUNCTION
```

## 4.12 Zusammenfassung der Anweisungen und Funktionen zur Zeichenkettenverarbeitung

Die folgende Tabelle führt die Anweisungen und Funktionen auf, die in BASIC zur Arbeit mit Zeichenketten zur Verfügung stehen.

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Teil einer Zeichenkette separieren | **LEFT$** | Gibt eine gegebene Anzahl von Zeichen von der linken Seite einer Zeichenkette zurück |
| | **RIGHT$** | Gibt eine gegebene Anzahl von Zeichen von der rechten Seite einer Zeichenkette zurück |
| | **LTRIM$** | Gibt eine Kopie einer Zeichenkette zurück, deren führende Leerzeichen entfernt sind |
| | **RTRIM$** | Gibt eine Kopie einer Zeichenkette zurück, deren nachfolgende Leerzeichen entfernt sind |
| | **MID$** (Funktion) | Gibt eine gegebene Anzahl von Zeichen ab einer beliebigen Zeichenkettenposition zurück |
| Zeichenketten durchsuchen | **INSTR** | Sucht nach einer Zeichenkette innerhalb einer anderen Zeichenkette |

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Umwandeln von Klein- in Großbuchstaben bzw. umgekehrt | **LCASE$** | Gibt die Kopie einer Zeichenkette zurück, in der alle Großbuchstaben (A - Z) in Kleinbuchstaben (a - z) umgewandelt sind; Kleinbuchstaben und andere Zeichen bleiben unverändert** |
| | **UCASE$** | Gibt die Kopie einer Zeichenkette zurück, in der alle Kleinbuchstaben (a - z) in Großbuchstaben (A - Z) umgewandelt sind; Großbuchstaben und andere Zeichen bleiben unverändert** |
| Zeichenketten verändern | **MID$** (Anweisung) | Ersetzt Teil einer Zeichenkette mit einer anderen Zeichenkette |
| | **LSET** | Ordnet eine Zeichenkette innerhalb einer Zeichenkette fester Länge linksbündig an |
| | **RSET** | Ordnet eine Zeichenkette innerhalb einer Zeichenkette fester Länge rechtsbündig an |
| Umwandeln von Zahlen in Zeichenketten und Zeichenketten in Zahlen | **STR$** | Gibt die Zeichenkettendarstellung des Wertes eines numerischen Ausdruckes zurück |
| | **VAL** | Gibt den numerischen Wert eines Zeichenkettenausdruckes zurück |
| Umwandeln von Zahlen in Datendateizeichenketten und Datendateizeichenketten in Zahlen* | **CV***Typ* | Wandelt in Programmen, die mit in älteren BASIC-Versionen erstellten Direktzugriffsdateien arbeiten, als Zeichenketten gespeicherte Zahlen in Microsoft Binärformatzahlen zurück |
| | **CV***Typ***MBF** | Wandelt Zahlen, die als Zeichenketten im Microsoft Binärformat gespeichert sind, in Zahlen im IEEE-Format um |

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| | **MK***Typ$* | Wandelt Microsoft Binärformatzahlen in Zeichenketten um, passend zur Speicherung in Direktzugriffsdateien, die mit älteren BASIC-Versionen erstellt wurden |
| | **MK***Typ***MBF$** | Wandelt Zahlen im IEEE-Format in Zeichenketten im Microsoft Binärformat um |
| Zeichenketten mit sich wiederholenden Zeichen bilden | **SPACE$** | Gibt eine aus Leerzeichen bestehende Zeichenkette zurück |
| | **STRING$** | Gibt eine aus einem einzigen wiederholten Zeichen bestehende Zeichenkette zurück |
| Die Länge einer Zeichenkette bekommen | **LEN** | Gibt an, wieviele Zeichen sich in einer Zeichenkette befinden |
| Arbeiten mit ASCII-Werten | **ASC** | Gibt den ASCII-Wert des angegebenen Zeichens zurück |
| | **CHR$** | Gibt das Zeichen mit dem angegebenen ASCII-Wert zurück |

* Für Sätze im Direktzugriff, deren Datenstruktur mit **TYPE...END TYPE** definiert ist, ist dies nicht länger erforderlich.

** Umlaute und "ß" bleiben ebenso unverändert.

# 5 Grafiken


5.1 Was Sie für grafische Programme benötigen 5.3

5.2 Bildpunkte und Bildschirmkoordinaten 5.5

5.3 Das Zeichnen von Grundformen: Punkte, Geraden, Rechtecke 5.7
- 5.3.1 Wie Sie Punkte mit **PSET** und **PRESET** grafisch darstellen 5.7
- 5.3.2 Wie Sie mit **LINE** Geraden und Rechtecke zeichnen 5.8
  - 5.3.2.1 Wie Sie die Option **STEP** einsetzen, um relative Koordinaten festzulegen 5.9
  - 5.3.2.2 Wie Sie Rechtecke zeichnen 5.11
  - 5.3.2.3 Wie Sie punktierte Geraden zeichnen 5.13

5.4 Wie Sie Kreise und Ellipsen mit **CIRCLE** zeichnen 5.14
- 5.4.1 Wie Sie Kreise zeichnen 5.14
- 5.4.2 Wie Sie Ellipsen zeichnen 5.15
- 5.4.3 Wie Sie Bögen zeichnen 5.17
- 5.4.4 Wie Sie Tortendiagramme zeichnen 5.21
- 5.4.5 Wie Sie Formen zeichnen, die mit dem Bildverhältnis anzupassen sind 5.22

5.5 Wie Sie ein grafisches Darstellungsfeld definieren 5.24

5.6 Wie Sie die Koordinaten eines Darstellungsfeldes mit **WINDOW** neu definieren 5.28
- 5.6.1 Die Reihenfolge von Koordinatenpaaren 5.32
- 5.6.2 Wie Sie sich logische und physikalische Koordinaten merken 5.33

5.7 Wie Sie Farben verwenden 5.34
- 5.7.1 Wie Sie eine Farbe für grafische Ausgabe wählen 5.35
- 5.7.2 Wie Sie die Vordergrund- oder Hintergrundfarbe verändern 5.36
- 5.7.3 Wie Sie Farben mit **PALETTE** und **PALETTE USING** verändern 5.39

5.8 Wie Sie Formen ausmalen 5.41
5.8.1 Ausmalen mit Farben 5.42
5.8.2 Zeichnen mit Mustern: Ausfüllen mit Mustern 5.44
5.8.2.1 Größe eines Kachelmusters in unterschiedlichen Bildschirmmodi 5.44
5.8.2.2 Wie Sie im Bildschirmmodus 2 ein einfarbiges Muster erzeugen 5.45
5.8.2.3 Wie Sie im Bildschirmmodus 1 ein vielfarbiges Muster erzeugen 5.48
5.8.2.4 Wie Sie im Bildschirmmodus 8 ein vielfarbiges Muster erzeugen 5.52

5.9 **DRAW**: Eine graphische Makro-Sprache 5.55

5.10 Grundlegende Techniken der Animation 5.58
5.10.1 Wie Sie Bilder mit **GET** speichern 5.58
5.10.2 Wie Sie Bilder mit **PUT** bewegen 5.61
5.10.3 Animation mit **GET** und **PUT** 5.65
5.10.4 Animation mit Bildschirmseiten 5.71

5.11 Beispielanwendungen 5.73
5.11.1 Balkendiagramm-Generator (BALKEN.BAS) 5.73
5.11.2 Mathematisch erzeugte Farben in einer Figur (MANDEL.BAS) 5.79
5.11.3 Editor für Muster (EDMUST.BAS) 5.86

5.12 Zusammenfassung der Grafikanweisungen und -funktionen 5.92

Dieses Kapitel zeigt Ihnen, wie Sie die Grafikanweisungen und -funktionen von BASIC verwenden, um auf Ihrem Bildschirm eine Vielzahl an Formen, Farben und Mustern zu erzeugen. Mit Grafik können Sie nahezu jedem Programm eine optische Dimension hinzufügen, ganz gleich, ob es sich um ein Spiel, ein Lernprogramm, eine wissenschaftliche Anwendung oder ein Finanzpaket handelt.

Wenn Sie dieses Kapitel durchgearbeitet haben, werden Sie wissen, wie die folgenden grafischen Aufgaben durchzuführen sind:

- Verwenden des physikalischen Koordinatensystems Ihres Personal Computer-Bildschirms, um einzelne Bildpunkte zu positionieren, diese Bildpunkte ein- oder auszuschalten sowie deren Farben zu verändern.
- Zeichnen von Geraden.
- Zeichnen und Ausfüllen einfacher Formen wie Kreise, Ovale und Rechtecke.
- Begrenzen des die grafische Ausgabe anzeigenden Bildschirmbereiches durch Verwendung von Darstellungsfeldern.
- Justieren der zum Positionieren eines Bildpunktes verwendeten Koordinaten durch Neudefinierung von Bildschirmkoordinaten.
- Verwendung von Farbe bei grafischer Ausgabe.
- Erstellen von Mustern sowie deren Verwendung zum Ausfüllen umrandeter Abbildungen.
- Kopieren von Bildern und deren Reproduktion an anderer Bildschirmstelle.
- Bewegte grafische Ausgabe.

***Wichtig*** Bevor Sie versuchen, eines der in diesem Kapitel aufgeführten Programme zu starten, lesen Sie bitte Abschnitt 5.1.

## 5.1 Was Sie für grafische Programme benötigen

Um die in diesem Kapitel gezeigten Beispiele zu starten, muß Ihr Computer entweder eingebaute Grafikfähigkeit oder eine grafische Adapterkarte besitzen, wie z. B. eine Farb-Grafikkarte (Color Graphics Adapter, CGA), einen Erweiterten Grafikadapter (Enhanced Graphics Adapter, EGA), einen Video Grafikadapter (VGA) oder einen Hercules Bildschirmadapter. Ebenso benötigen Sie einen Bildschirm (entweder monochrom oder farbig), der aus Bildpunkten aufgebaute Grafiken unterstützt.

Behalten Sie darüber hinaus bitte im Auge, daß alle diese Programme einen die grafische Ausgabe unterstützenden Bildschirmmodus erfordern. (Der Bildschirmmodus steuert die Schärfe grafischer Bilder, die Anzahl der verfügbaren Farben und den Teil des anzuzeigenden Bildschirmspeichers.) Um einen grafischen Ausgabemodus auszuwählen, verwenden Sie in Ihrem Programm die folgende Anweisung vor irgendeiner der in der Tabelle in Abschnitt 5.12 aufgeführten Anweisungen oder Funktionen:

**SCREEN** *Modus*

*Modus* kann hier abhängig von der Monitor/Adapterkombination, die Sie in Ihrem Computer installiert haben, entweder 1, 2, 7, 8, 9, 10, 11, 12 oder 13 sein.

Wenn Sie nicht sicher sind, ob die Benutzer Ihrer Programme grafikunterstützende Hardware haben oder nicht, können Sie die folgende einfache Prüfung durchführen:

```
CONST FALSCH = 0, WAHR = NOT FALSCH
' Test zur Sicherstellung, daß der Benutzer Hardware
' mit Farb/Grafikfähigkeit hat:
ON ERROR GOTO Meld  ' Schalte Fehlerverfolgung ein.
SCREEN 1            ' Versuche grafischen Modus eins.
ON ERROR GOTO 0      ' Schalte Fehlerverfolgung aus.
IF KeineGraf THEN END       ' Beende, wenn keine
                            ' grafische Hardware.
.
.
.
END
' Fehlerbehandlungsroutine:
Meld:
   PRINT "Dieses Programm erfordert Grafikfähigkeit."
   KeineGraf = WAHR
   RESUME
```

Wenn der Benutzer lediglich einen monochromen Bildschirm ohne Grafikadapter hat, ergibt die Anweisung **SCREEN** einen Fehler, als dessen Folge eine Verzweigung in die Fehlerbehandlungsroutine `Meld` ausgelöst wird. (Weitere Informationen zur Fehlerverfolgung finden Sie in Kapitel 6, "Fehler- und Ereignisverfolgung".)

## 5.2 Bildpunkte und Bildschirmkoordinaten

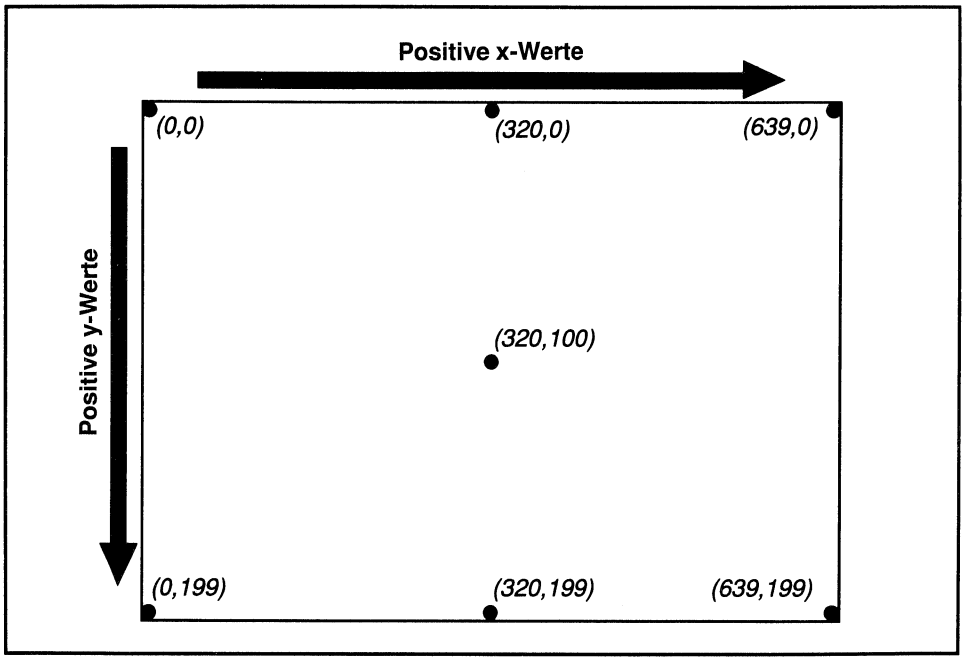
Auf einem Bildschirm sind Formen und Abbildungen aus einzelnen Lichtpunkten zusammengesetzt, die als Bildelemente oder als "Bildpunkte" (oder manchmal als "Pixel") bezeichnet werden. BASIC zeichnet und malt auf den Bildschirm, indem es diese Bildpunkte ein- oder ausschaltet sowie deren Farben verändert.

Ein typischer Bildschirm ist aus einem Raster von Bildpunkten zusammengesetzt. Die genaue Anzahl der Bildpunkte dieses Rasters hängt von der installierten Hardware und dem von Ihnen in der Anweisung **SCREEN** gewählten Bildschirmmodus ab. Je größer die Anzahl an einschaltbaren Bildpunkten ist, desto größer ist die Schärfe grafischer Ausgabe. Eine Anweisung `SCREEN 1` ergibt z. B. eine Auflösung von 320 Bildpunkten horizontal mit 200 Punkten vertikal (320 x 200 Bildpunkte), während eine Anweisung `SCREEN 2` eine Auflösung von 640 x 200 Bildpunkten ergibt. Die höhere horizontale Dichte im Bildschirmmodus 2 – 640 Bildpunkte pro Zeile gegenüber 320 Bildpunkten pro Zeile – gibt Bildern gegenüber Bildschirmmodus 1 ein schärferes und weniger ausgefranstes Erscheinungsbild.

Abhängig von der Grafikfähigkeit Ihres Systems können Sie einen anderen Bildschirmmodus verwenden, der sogar höhere Auflösungen unterstützt (sowie weitere Bildschirmcharakteristika anpaßt). Weitere Informationen zu der Hardware und den Bildschirmmodi, die unterschiedliche Bildschirmauflösungen unterstützen, finden Sie in der Beschreibung der Anweisung **SCREEN** im *BASIC-Befehlsverzeichnis*.

Wenn Ihr Bildschirm einen der erforderlichen grafischen Modi hat, können Sie jeden Bildpunkt mit Hilfe von Koordinatenpaaren positionieren. Die erste Zahl in jedem Koordinatenpaar gibt die Bildpunktzahl ab der linken Seite des Bildschirms an, während die zweite Zahl in jedem Paar die Bildpunktzahl ab dem oberen Bildschirmrand angibt. Im Bildschirmmodus 2 hat der Bildpunkt in der äußerst oberen linken Ecke des Bildschirms zum Beispiel die Koordinaten (0,0), der Punkt in Bildschirmmitte hat die Koordinaten (320,100) und der Punkt in der äußerst unteren rechten Ecke des Bildschirms hat die Koordinaten (639,199), wie in Abbildung 5.1 gezeigt.

*Abbildung 5.1 Koordinaten ausgewählter Bildpunkte im Bildschirmmodus 2*




BASIC verwendet diese Bildschirmkoordinaten zur Bestimmung, wo Grafiken angezeigt werden (zum Beispiel die Endpunkte einer Geraden oder der Mittelpunkt eines Kreises), wie in Abschnitt 5.3, "Das Zeichnen von Grundformen" gezeigt wird.

***Hinweis*** Beachten Sie die Unterschiede zwischen grafischen Koordinaten und Koordinaten im Textmodus in einer **LOCATE**-Anweisung. Zum einen ist **LOCATE** nicht so genau: Grafische Koordinaten zeigen auf einzelne Bildpunkte des Bildschirms, während von **LOCATE** verwendete Koordinaten Positionen von Zeichen sind. Zum anderen werden Koordinaten im Textmodus in der Form *Zeile*, *Spalte* angegeben, wie im folgenden Beispiel:

```
' Spring in die 10-te Zeile, 15-te Spalte, gib
' dann die gezeigte Meldung aus:
LOCATE 10, 15
PRINT "Dies sollte in der Bildschirmmitte
erscheinen."
```

Dies ist die Umkehrung der grafischen Koordinaten, die in der Form *Spalte, Zeile* angegeben werden.

Eine **LOCATE**-Anweisung hat keine Auswirkung auf die Positionierung grafischer Ausgaben auf dem Bildschirm. Weitere Erläuterungen zu Textkoordinaten und wie diese verwendet werden können, um die Position geschriebener Ausgabe auf dem Bildschirm zu steuern, finden Sie in Kapitel 3 dieses Handbuches, "Datei- und Geräte-E/A", und in der Beschreibung zu der **LOCATE**-Anweisung im *BASIC-Befehlsverzeichnis*.

## 5.3 Das Zeichnen von Grundformen: Punkte, Geraden, Rechtecke

Sie können Koordinatenwerte an BASIC-Grafikanweisungen übergeben, um eine Vielzahl einfacher Abbildungen zu erstellen, wie in den Abschnitten 5.3.1 bis 5.3.2 gezeigt wird.

### 5.3.1 Wie Sie Punkte mit PSET und PRESET grafisch darstellen

Die unterste Ebene der Steuerung grafischer Ausgabe ist das einfache Ein- und Ausschalten einzelner Bildpunkte. In BASIC erledigen Sie dies mit den Anweisungen **PSET** (für Pixel-SET) und **PRESET** (für Pixel-RESET). Daher gibt die Anweisung

**PSET** (*x*,*y*)

dem Bildpunkt mit den Koordinaten (*x*,*y*) die aktuelle Vordergrundfarbe.

Die Anweisung

**PRESET** (*x*,*y*)

gibt dem Bildpunkt mit den Koordinaten (*x*,*y*) die aktuelle Hintergrundfarbe.

Auf monochromen Monitoren ist die Vordergrundfarbe diejenige Farbe, die für geschriebenen Text verwendet wird; sie ist normalerweise weiß, amber oder hellgrün; die Hintergrundfarbe auf einem monochromen Monitor ist normalerweise schwarz oder dunkelgrün. Mit dem wahlweisen Argument *Farbe* können Sie eine andere Farbe auswählen, mit der **PSET** und **PRESET** arbeiten:

**PSET** (*x*,*y*), *Farbe*

**PRESET** (*x*,*y*), *Farbe*

Weitere Informationen zur Auswahl von Farben finden Sie in Abschnitt 5.7.

Da **PSET** standardmäßig die aktuelle Vordergrundfarbe verwendet, und **PRESET** standardmäßig die aktuelle Hintergrundfarbe verwendet, löscht **PRESET** ohne ein Farbargument einen mit **PSET** grafisch dargestellten Punkt, wie im nächsten Beispiel gezeigt:

```
SCREEN 2        ' Auflösung 640 x 200
PRINT "Beliebige Taste zum Beenden."
DO
   ' Zeichne eine horizontale Gerade von ganz links
   ' bis ganz rechts:
   FOR X = 0 TO 640
      PSET (X,100)
   NEXT
   ' Lösche die Gerade von rechts nach links:
   FOR X = 640 TO 0 STEP -1
      PRESET (X,100)
   NEXT
LOOP UNTIL INKEY$ <> ""
END
```

Es ist zwar möglich, jede beliebige Abbildung nur mit **PSET**-Anweisungen durch Manipulation einzelner Bildpunkte zu zeichnen, die Ausgabe neigt jedoch dazu, ziemlich langsam zu sein, da die meisten Bilder aus vielen Bildpunkten bestehen. BASIC hat mehrere Anweisungen, die die Geschwindigkeit, mit der einfache Abbildungen – wie Geraden, Rechtecke und Ellipsen – gezeichnet werden, drastisch erhöhen, wie in Abschnitt 5.3.2 und 5.4.1 bis 5.4.4 gezeigt.

## 5.3.2 Wie Sie mit LINE Geraden und Rechtecke zeichnen

Bei der Verwendung von **PSET** oder **PRESET** geben Sie nur ein Koordinatenpaar an, da Sie nur mit einem Punkt auf dem Bildschirm arbeiten. Mit **LINE** geben Sie zwei Paare an, eines für jedes Ende eines Geradenabschnittes. Die einfachste Form der **LINE**-Anweisung lautet wie folgt:

**LINE** (*x1,y1*) - (*x2,y2*)

wobei (*x1,y1*) die Koordinaten des einen Endes des Geradenabschnittes sind, und (*x2,y2*) die Koordinaten des anderen Endes.

Die folgende Anweisung zeichnet zum Beispiel eine gerade Linie vom Bildpunkt mit den Koordinaten (10,10) zu dem Bildpunkt mit den Koordinaten (150,200):

```
SCREEN 1
LINE (10,10) - (150,200)
```

Beachten Sie, daß BASIC die Reihenfolge der Koordinatenpaare nicht beachtet: eine Gerade von (*x1,y1*) nach (*x2,y2*) ist dasselbe wie eine Gerade von (*x2,y2*) nach (*x1,y1*). Daher könnten Sie das vorhergehende Beispiel auch schreiben als:

```
SCREEN 1
LINE (150,200) - (10,10)
```

Die Umkehrung der Koordinatenreihenfolge könnte jedoch eine Auswirkung auf nachfolgende Grafikanweisungen haben, wie im nächsten Abschnitt gezeigt:

#### 5.3.2.1 Wie Sie die Option STEP einsetzen, um relative Koordinaten festzulegen

Bis zu diesem Punkt sind Bildschirmkoordinaten als absolute Werte vorgestellt worden, die die horizontalen und vertikalen Abstände von der äußerst linken oberen Bildschirmecke angeben, deren Koordinaten (0,0) sind. Mit der Verwendung der Option **STEP** in jeder der folgenden Grafikanweisungen können Sie die auf **STEP** folgenden Koordinaten jedoch relativ zum letzten Bildschirmpunkt, auf den Bezug genommen wurde, machen:

**CIRCLE**
**GET**
**LINE**
**PAINT**
**PRESET**
**PSET**
**PUT**

Wenn Sie sich Bilder so vorstellen, als seien sie auf dem Bildschirm mit einem dünnen Pinsel gezeichnet, der exakt die Größe eines Bildpunktes hat, dann ist der letzte Punkt, auf den Bezug genommen wird, die Position dieses Pinsels oder "grafischen Cursors", wenn dieser das Zeichnen eines Bildes beendet hat.

Zum Beispiel belassen die folgenden Anweisungen den grafischen Cursor auf dem Bildpunkt mit den Koordinaten (100,150):

```
SCREEN 2
LINE (10,10) - (100,150)
```

Wenn in dem Programm die nächste Grafikanweisung

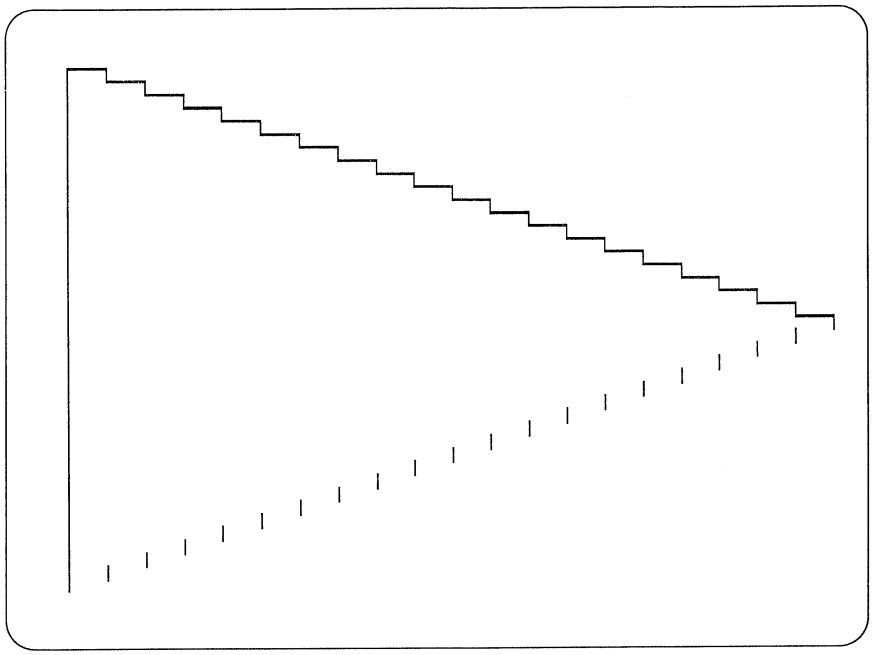
```
PSET STEP (20,20)
```

ist, dann befindet sich der von **PSET** gezeichnete Punkt nicht in dem oberen linken Quadranten des Bildschirms. Stattdessen hat die Option **STEP** die Koordinaten (20,20) relativ zu dem letzten angesprochenen Punkt, der die Koordinaten (100,150) hat, gemacht. Dies gibt dem Punkt die absoluten Koordinaten (100+20,150+20) oder (120,170).

Im letzten Beispiel wird der letzte angesprochene Punkt von einer vorhergehenden Grafikanweisung bestimmt. Sie können einen Bezugspunkt auch innerhalb derselben Anweisung bestimmen, wie im nächsten Beispiel gezeigt:

```
' Setze die Auflösung auf 640 x 200 Bildpunkte und
' mache die Bildschirmmitte zum letzten Bezugspunkt:
SCREEN 2
' Zeichne eine Gerade von der linken unteren Ecke des
' Bildschirms zur oberen linken Ecke:
LINE STEP(-310, 100)-STEP(0, -200)
' Zeichne die "Treppenstufen" abwärts von der oberen
' linken Ecke zur rechten Seite des Bildschirms:
FOR I% = 1 TO 20
   LINE -STEP(30, 0)     ' Zeichne die horizontale
                         ' Gerade.
   LINE -STEP(0, 5)      ' Zeichne die vertikale
NEXT                     ' Gerade.
' Zeichne die unverbundenen vertikalen Geradenstücke
' von der rechten Seite zur unteren linken Ecke:
FOR I% = 1 TO 20
   LINE STEP(-30, 0)-STEP(0, 5)
NEXT
DO: LOOP WHILE INKEY$ = ""     ' Warte auf
                               ' Tastenbetätigung.
```

***Ausgabe***



***Hinweis*** Beachten Sie die leere **DO**-Schleife am Ende des letzten Programmes. Wenn Sie ein kompiliertes selbständiges BASIC-Programm starten, das grafische Ausgabe erzeugt, benötigt Ihr Programm einen Mechanismus wie diesen am Ende, um die Ausgabe auf dem Bildschirm zu halten. Andernfalls verschwindet diese vom Bildschirm, bevor der Benutzer Zeit hat, sie zu betrachten.

### 5.3.2.2 Wie Sie Rechtecke zeichnen

Mit Verwendung der bereits dargestellten Formen der **LINE**-Anweisung ist es ziemlich einfach, ein kurzes Programm zu schreiben, das vier gerade Linien zur Bildung eines Rechteckes verbindet, wie hier gezeigt:

```
SCREEN 1    ' Auflösung 320 x 200 Bildpunkte
' Zeichne ein Rechteck, das auf einer Seite 120
' Bildpunkte mißt:
LINE (50,50) - (170,50)
LINE - STEP (0,120)
LINE - STEP (-120,0)
LINE - STEP (0,-120)
```

BASIC unterstützt jedoch einen einfacheren Weg, Rechtecke zu zeichnen, indem Sie eine einzelne **LINE**-Anweisung mit der Option **B** (für Box) verwenden. Die Option **B** wird im nächsten Beispiel, das dieselbe Ausgabe wie die vier **LINE**-Anweisungen im vorhergehenden Programm erzeugt, gezeigt:

```
SCREEN 1    ' Auflösung 320 x 200 Bildpunkte
' Zeichne ein Rechteck mit den Koordinaten
' (50,50) für die obere linke Ecke, und
' (170,170) für die untere rechte Ecke:
LINE (50,50) - (170,170),,B
```

Wenn Sie die Option **B** hinzufügen, verbindet die **LINE**-Anweisung nicht länger die angegebenen zwei Punkte mit einer geraden Linie; stattdessen verwendet sie diese Punkte als gegenüberliegende Ecken (obere linke und untere rechte) eines Rechtecks und zeichnet anschließend das Rechteck.

Beachten Sie, daß im letzten Beispiel vor dem **B** zwei Kommata stehen. Das erste Komma steht hier als Platzhalter für eine nicht verwendete Option (das Argument *Farbe*), die es Ihnen erlaubt, die Farbe einer Zeile oder die Farbe der Seiten eines Rechtecks auszusuchen. (Weitere Informationen über die Verwendung von Farben finden Sie in Abschnitt 5.7.)

Wie schon zuvor ist die Reihenfolge, in der die Koordinatenpaare angegeben werden, bedeutungslos, so daß das Rechteck des letzten Beispiels ebenso mit dieser Anweisung gezeichnet werden könnte:

```
LINE (170,170) - (50,50),,B
```

Das Hinzufügen der Option **F** (für Fill) nach **B** malt das Innere des Rechtecks mit derselben Farbe aus, die zum Zeichnen der Seiten verwendet wird. Mit einem monochromen Bildschirm ist diese die gleiche wie die für geschriebenen Text verwendete Vordergrundfarbe. Wenn Ihre Hardware Farbfähigkeiten besitzt, können Sie diese Farbe mit dem wahlfreien Argument *Farbe* verändern (siehe auch Abschnitt 5.7.1, "Wie Sie eine Farbe für grafische Ausgabe wählen").

Die hier eingeführte Schreibweise zum Zeichnen eines Rechtecks ist die allgemein in BASIC verwendete Schreibweise, um einen rechtwinkeligen Grafikbereich zu definieren, und sie erscheint ebenfalls in den Anweisungen **GET** und **VIEW**.

{**GET** | **LINE** | **VIEW**} (*x1*,*y1*) - (*x2*,*y2*),...

Hier sind (*x1*,*y1*) und (*x2*,*y2*) die Koordinaten sich diagonal gegenüberliegender Ecken des Rechtecks (oben links und unten rechts). (Erläuterungen zu **VIEW** finden Sie in Abschnitt 5.5, "Wie Sie ein grafisches Darstellungsfeld definieren", und Informationen zu **GET** und **PUT** finden Sie in Abschnitt 5.10, "Grundlegende Techniken der Animation".)

### 5.3.2.3 Wie Sie punktierte Geraden zeichnen

Die vorhergehenden Abschnitte erläutern die Verwendung von **LINE**, um durchgezogene Geraden zu zeichnen und diese für Rechtecke zu verwenden; das heißt, es werden keine Bildpunkte ausgelassen. Unter Verwendung einer anderen Option mit **LINE** können Sie stattdessen punktierte oder unterbrochene Geraden zeichnen. Dieses Verfahren wird als "Geradengestaltung" bezeichnet. Es folgt die Schreibweise, mit der unter Verwendung der aktuellen Vordergrundfarbe eine unterbrochene Gerade vom Punkt (*x1,y1*) zum Punkt (*x2,y2*) gezeichnet wird:

**LINE** (*x1,y1*) - (*x2,y2*), , [B], *Struktur*

Hier ist *Struktur* eine 16-Bit dezimale oder hexadezimale Ganzzahl. (In der Beschreibung zu der Anweisung **LINE** im *BASIC-Befehlsverzeichnis* finden Sie die komplette allgemeine Schreibweise für **LINE**.)

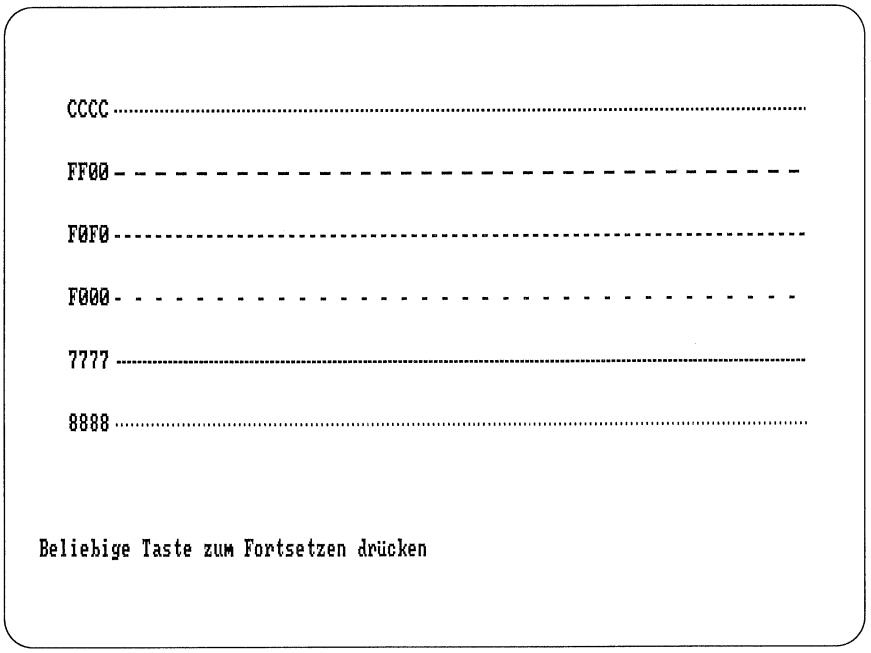
**LINE** verwendet die binäre Darstellung des Arguments der Geradengestaltung, um Striche und Leerzeichen zu erzeugen, wobei ein 1-Bit bedeutet "schalte den Bildpunkt ein" und ein 0-Bit bedeutet "lasse den Bildpunkt ausgeschaltet". Zum Beispiel ist die hexadezimale Ganzzahl &HCCCC gleich der binären Ganzzahl 1100110011001100; bei Verwendung als *Struktur*-Argument zeichnet sie eine Gerade mit abwechselnd zwei Bildpunkten ein, zwei Bildpunkten aus.

#### Beispiel

Das folgende Beispiel zeigt unterschiedlich unterbrochene Geraden, die mit unterschiedlichen Werten für *Struktur* erzeugt werden:

```
SCREEN 2       ' Auflösung 640 x 200 Bildpunkte
' Daten für Struktur:
DATA &HCCCC, &HFF00, &HF0F0
DATA &HF000, &H7777, &H8888
Zeile%   = 4
Spalte%  = 4
XLinks%  = 60
XRechts% = 600
Y%       = 28
FOR I% = 1 TO 6
   READ Struktur%
   LOCATE Zeile%, Spalte%
   PRINT HEX$(Struktur%)
   LINE (XLinks%,Y%) - (XRechts%,Y%),,,Struktur%
   Zeile% = Zeile% + 3
   Y%     = Y% + 24
NEXT
```

***Ausgabe***




## 5.4 Wie Sie Kreise und Ellipsen mit CIRCLE zeichnen

Die Anweisung **CIRCLE** zeichnet eine Vielfalt kreisförmiger, elliptischer oder ovaler Formen. Außerdem zeichnet **CIRCLE** Bögen (Ausschnitte von Kreisen) sowie Keile, die den Stücken einer Torte gleichen. Kurz gesagt, im Grafikmodus können Sie fast alle Kurvenarten mit einer Abwandlung von **CIRCLE** erstellen.

### 5.4.1 Wie Sie Kreise zeichnen

Um einen Kreis von Hand zu zeichnen, müssen Sie nur zwei Dinge wissen: die Position seines Mittelpunkts sowie die Länge seines Radius (der Abstand vom Mittelpunkt zu einem beliebigen Punkt auf dem Kreis). Diese Informationen (und ein Zirkel) sind zum Zeichnen eines Kreises ausreichend.

Ebenso benötigt BASIC nur die Position des Kreismittelpunktes sowie die Länge des Radius, um einen Kreis zu zeichnen. Die einfachste Form der Schreibweise für **CIRCLE** ist

**CIRCLE** [**STEP**] (*x,y*), *Radius*

wobei *x,y* die Koordinaten des Mittelpunktes sind, und *Radius* der Radius des Kreises ist. Das nächste Beispiel zeichnet einen Kreis mit dem Mittelpunkt (200,200) und einem Radius von 75:

```
SCREEN 2
CIRCLE (200,100), 75
```

Beachten Sie, daß Sie das vorhergehende Beispiel wie folgt neuschreiben könnten, indem Sie die **STEP**-Option verwenden, um die Koordinaten relativ zum Mittelpunkt zu machen und nicht zur oberen linken Ecke:

```
SCREEN 2        ' Macht Mittelpunkt des Bildschirms
                ' (320,100) zum Bezugspunkt für die
                ' CIRCLE-Anweisung
CIRCLE STEP (-120,0), 75
```

## 5.4.2 Wie Sie Ellipsen zeichnen

Solange Sie Kreise zeichnen, stellt die **CIRCLE**-Anweisung einen Wert ein, der als "Seitenverhältnis" (Aspect Ratio) bezeichnet wird, um sicherzustellen, daß der Kreis rund erscheint und nicht ellipsenförmig. Dennoch müssen Sie vielleicht das Seitenverhältnis anpassen, um Kreise auf Ihrem Bildschirm richtig darzustellen. Sie können das Seitenverhältnis verändern, um eine ovale Abbildung, die als Ellipse bezeichnet wird, auf dem Bildschirm darzustellen. Verwenden Sie in beiden Fällen folgende Schreibweise

**CIRCLE** [**STEP**] (*x,y*), *Radius*, , , *Aspekt*

wobei *Aspekt* eine positive reelle Zahl ist. (Weitere Informationen zu dem Seitenverhältnis und wie es für verschiedene Bildschirmmodi zu berechnen ist, finden Sie in Abschnitt 5.4.5.)

Machen Sie sich im Moment keine Gedanken zu den zusätzlichen Kommata zwischen *Radius* und *Aspekt*; sie sind Platzhalter für andere Optionen, die **CIRCLE** angeben, welche Farbe zu verwenden ist (wenn Sie einen Farbmonitoradapter haben und einen der Bildschirmmodi verwenden, die Farbe unterstützen), oder ob ein Bogen oder Keil zu zeichnen ist (weitere Informationen zu diesen Optionen finden Sie in den Abschnitten 5.4.3, "Wie Sie Bögen zeichnen" und 5.7.1,"Wie Sie eine Farbe für grafische Ausgabe wählen").

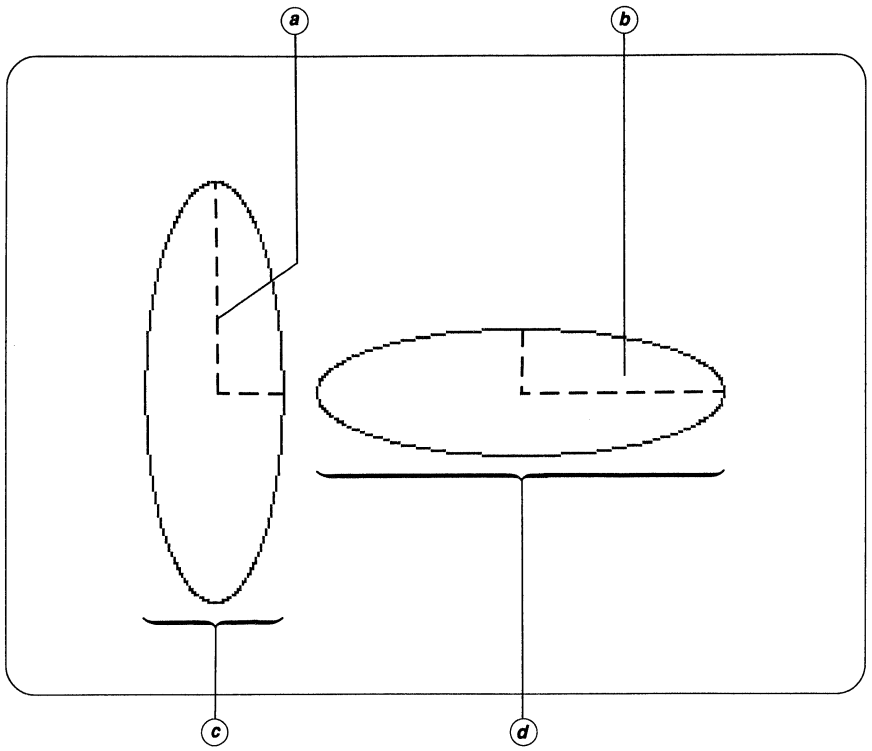
Da das Argument *Aspekt* das Verhältnis der vertikalen zur horizontalen Dimension angibt, führen große Werte für *Aspekt* zu Ellipsen, die sich entlang der vertikalen Achse erstrecken, während kleine Werte für *Aspekt* zu Ellipsen führen, die sich entlang der horizontalen Achse erstrecken. Da eine Ellipse zwei Radien hat – einen horizontalen x-Radius und einen vertikalen y-Radius (siehe folgende Abbildung) – verwendet BASIC das einzelne Argument *Radius* in einer **CIRCLE**-Anweisung wie folgt: wenn *Aspekt* kleiner als 1 ist, ist *Radius* der x-Radius; wenn *Aspekt* größer gleich 1 ist, ist *Radius* der y-Radius.

### Beispiel

Das folgende Beispiel und seine Ausgabe zeigen, wie unterschiedliche Werte von *Aspekt* sich darauf auswirken, ob die **CIRCLE**-Anweisung das Argument *Radius* als den x-Radius oder den y-Radius einer Ellipse verwendet.

```
SCREEN 1
' Dies zeichnet die linke Ellipse:
CIRCLE (60, 100), 80, , , , 3
' Dies zeichnet die rechte Ellipse:
CIRCLE (180, 100), 80, , , , 3/10
```

***Ausgabe***




*a) Radius*
*b) Radius*
*c) In der **CIRCLE**-Anweisung ist das Seitenverhältnis größer als 1*
*d) In der **CIRCLE**-Anweisung ist das Seitenverhältnis kleiner als 1*

## 5.4.3 Wie Sie Bögen zeichnen

Ein Bogen ist ein Ausschnitt einer Ellipse; mit anderen Worten, eine kurze, gekrümmte Linie. Um zu verstehen, wie die **CIRCLE**-Anweisung einen Bogen zeichnet, müssen Sie wissen, wie BASIC Winkel mißt.

BASIC verwendet Bogenmaß (Radian) als seine Einheit zur Winkelmessung, jedoch nicht nur in der Anweisung **CIRCLE**, sondern auch in eingebauten trigonometrischen Funktionen wie **COS**, **SIN** oder **TAN**. (Die einzige Ausnahme zu dieser Verwendung von Bogenmaß ist die Anweisung **DRAW**, die Winkelmessungen in Grad erwartet. Weitere Informationen zu **DRAW** finden Sie in Abschnitt 5.9.)

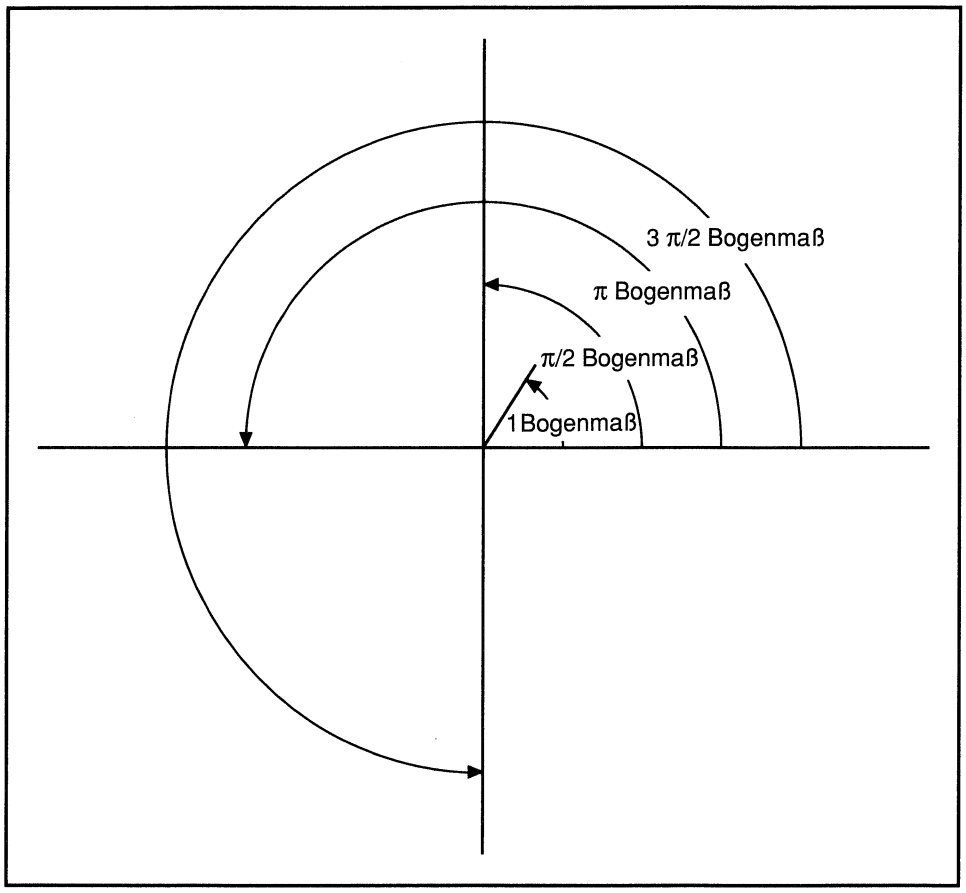
Bogenmaß (Radian) ist eng verknüpft mit dem Radius eines Kreises. Tatsächlich ist das Wort "Radian" abgeleitet von dem Wort "Radius". Der Umfang eines Kreises ist gleich 2x$\pi$x*Radius*, wobei $\pi$ ungefähr 3,14159265 entspricht. Die Anzahl der Radianten in einer kompletten Winkeldrehung (oder 360°) ist gleich 2x$\pi$ oder etwas mehr als 6,28. Wenn Sie es gewohnt sind, Winkel in Grad anzugeben, sind unten einige der wichtigsten Umrechnungen.

| ***Winkel in Grad*** | ***Winkel in Bogenmaß*** |
|---|---|
| 360 | $2\pi$ (ungefähr 6,283) |
| 180 | $\pi$ (ungefähr 3,142) |
| 90 | $\pi/2$ (ungefähr 1,571) |
| 60 | $\pi/3$ (ungefähr 1,047) |

Die allgemeine Formel zur Umwandlung von Grad in Bogenmaß ergibt sich aus der Multiplikation von Grad mit $\pi/180$.

Stellen Sie sich ein Ziffernblatt auf dem Bildschirm vor: **CIRCLE** mißt den Winkel beginnend bei der Position "3.00 Uhr", drehend entgegen dem Uhrzeigersinn, wie in Abbildung 5.2 gezeigt.

*Abbildung 5.2 Wie Winkel für **CIRCLE** gemessen werden*




Um einen Bogen zu zeichnen, geben Sie die Winkelargumente an, die die Grenzen des Bogens definieren:

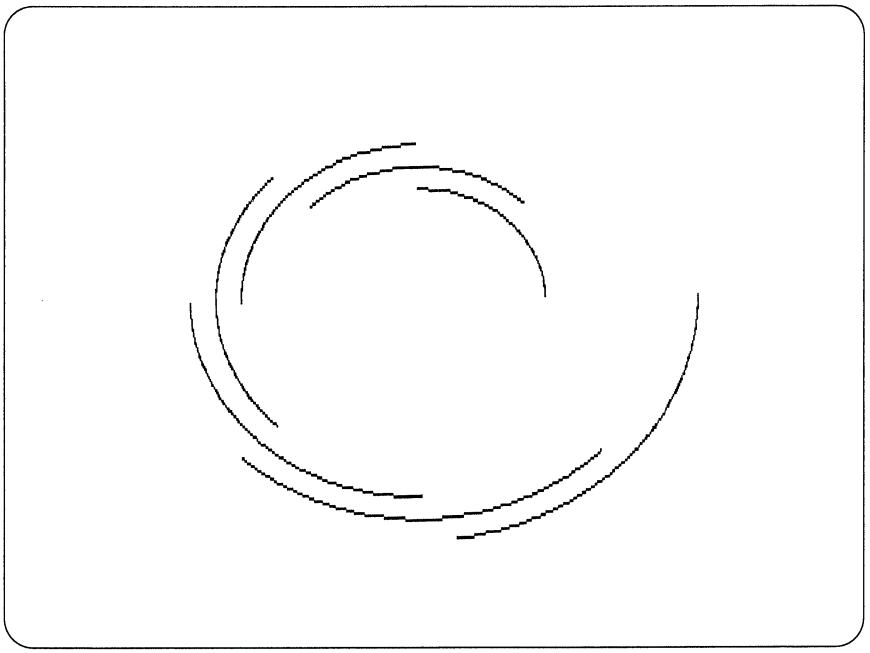
**CIRCLE** [**STEP**] (*x*,*y*), *Radius*, [*Farbe*], *Beginn*, *Ende* [, *Aspekt*]

### Beispiel

Die **CIRCLE**-Anweisungen des nächsten Beispiels zeichnen sieben Bögen, wobei der innerste Bogen bei der Position "3.00 Uhr" (0 Bogenmaß) und der äußerste Bogen bei der Position "6.00 Uhr" ($3\pi/2$ Bogenmaß) beginnt, wie Sie in der Ausgabe sehen können:

```
SCREEN 2
CLS
CONST PI = 3.141592653589#
StartWinkel = 0
FOR Radius% = 100 TO 220 STEP 20
   EndWinkel = StartWinkel + (PI / 2.01)
   CIRCLE (320, 100),Radius%, , StartWinkel, EndWinkel
   StartWinkel = StartWinkel + (PI / 4)
NEXT Radius%
```

### *Ausgabe*



## 5.4.4 Wie Sie Tortendiagramme zeichnen

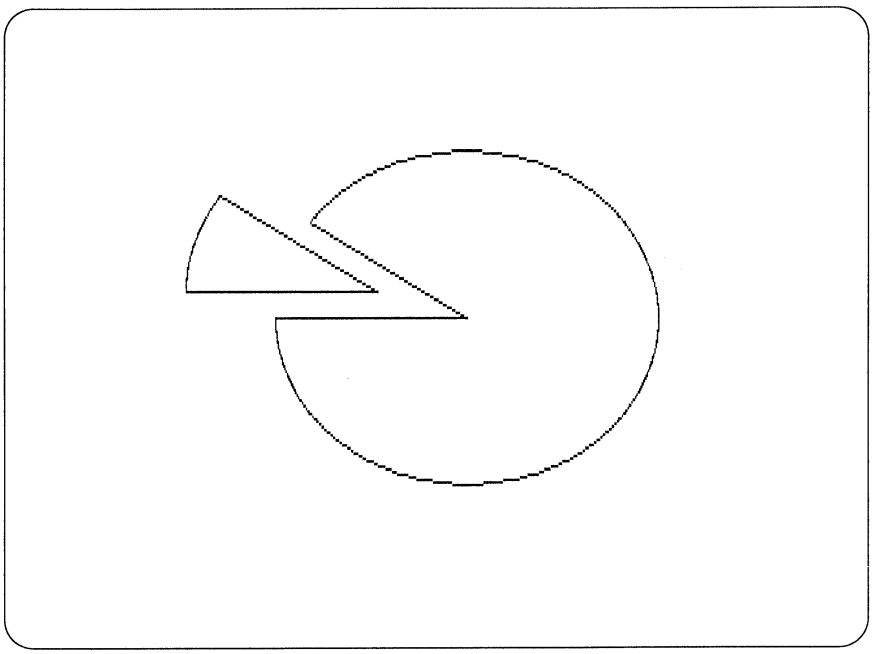
Wenn Sie für **CIRCLE** entweder das Argument *Beginn* oder das Argument *Ende* negativ angeben, können Sie den Bogen an seinem Anfangs- bzw. Endpunkt mit dem Kreismittelpunkt verbinden. Wenn Sie beide Argumente negativ angeben, können Sie Formen zeichnen, die von einem Keil, der einem Stück Torte gleicht, bis zu der Torte selbst (ohne das fehlende Stück) reichen.

### Beispiel

Dieses Beispiel zeichnet ein Tortendiagramm, dem ein Stück fehlt.

```
SCREEN 2
CONST RADIUS = 150, PI = 3.141592653589#
StartWinkel = 2.5
EndWinkel = PI
' Zeichne den Keil:
CIRCLE (320, 100), RADIUS, , -StartWinkel, -EndWinkel
' Vertausche die Werte für den Start- und Endwinkel:
SWAP StartWinkel, EndWinkel
' Bewege den Mittelpunkt 10 Bildpunkte nach unten und
' 70 Bildpunkte nach rechts, zeichne dann die Torte,
' der der Keil fehlt:
CIRCLE STEP(70,10), RADIUS, , -StartWinkel, -EndWinkel
```

***Ausgabe***



## 5.4.5 Wie Sie Formen zeichnen, die mit dem Seitenverhältnis anzupassen sind

Wie in Abschnitt 5.4.2, "Wie Sie Ellipsen zeichnen", erläutert, korrigiert die BASIC-Anweisung **CIRCLE** automatisch das Seitenverhältnis, das angibt, wie Abbildungen auf dem Bildschirm skaliert werden. Mit anderen Grafikanweisungen müssen Sie jedoch horizontale und vertikale Dimensionen selbst skalieren, damit Formen im richtigen Größenverhältnis erscheinen. Obwohl die folgende Anweisung zum Beispiel ein Rechteck zeichnet, das auf allen Seiten 100 Bildpunkte mißt, sieht dieses Rechteck nicht wie ein Quadrat aus:

```
SCREEN 1
LINE (0, 0) - (100, 100), ,B
```

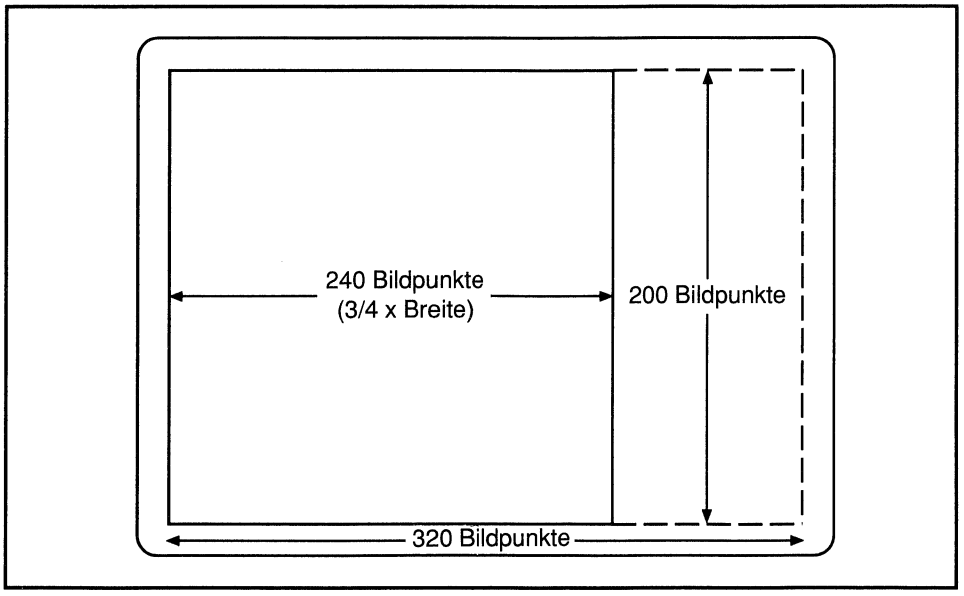
Dies ist tatsächlich keine optische Täuschung; das Rechteck ist wirklich höher als es breit ist. Dies rührt daher, daß im Bildschirmmodus 1 mehr Platz zwischen vertikalen als zwischen horizontalen Bildpunkten gelassen wird. Um ein richtiges Quadrat zu zeichnen, müssen Sie das Seitenverhältnis ändern.

Das Seitenverhältnis ist wie folgt definiert: betrachten Sie in einem gegebenen Bildschirmmodus zwei Geraden, eine vertikale und eine horizontale, die mit gleicher Länge erscheinen. Das Seitenverhältnis ist die Anzahl von Bildpunkten auf der vertikalen Geraden, dividiert durch die Anzahl der Bildpunkte auf der horizontalen Geraden. Dieses Verhältnis ist abhängig von zwei Faktoren:

1. Aufgrund der Art, wie Bildpunkte auf den meisten Bildschirmen in Abständen angeordnet werden, hat eine horizontale Zeile in allen Bildschirmmodi, außer in den Modi 11 und 12, mehr Bildpunkte als eine vertikale Spalte mit der exakt gleichen physikalischen Länge.
2. Der Standardbildschirm eines Personal Computers ist breiter als er hoch ist. Normalerweise ist das Verhältnis von Bildschirmhöhe zu Bildschirmbreite gleich 3:4.

Um zu sehen, wie diese beiden Faktoren bei der Aufstellung des Seitenverhältnisses zusammenwirken, betrachten Sie einen Bildschirm nach einer Anweisung **SCREEN 1**, die eine Auflösung von 320 x 200 Bildpunkten angibt. Wenn Sie ein Rechteck vom oberen bis zum unteren Bildschirmrand und dreiviertel des Weges von der linken bis zur gegenüberliegenden Bildschirmseite zeichnen, erhalten Sie ein Quadrat, wie in Abbildung 5.3 gezeigt.

*Abbildung 5.3 Das Seitenverhältnis im Bildschirmmodus 1*



Wie Sie dem Diagramm entnehmen können, hat dieses Quadrat eine Höhe von 200 Bildpunkten und eine Breite von 240 Bildpunkten. Das Verhältnis der Quadrathöhe zu seiner Breite (200/240, oder einfacher 5/6) ist das Seitenverhältnis für diese Bildschirmauflösung. Mit anderen Worten, um ein Quadrat bei einer Auflösung von 320 x 200 zu zeichnen, müssen Sie seine Höhe in Bildpunkten gleich 5/6 mal seiner Breite in Bildpunkten angeben, wie in dem nächsten Beispiel gezeigt:

```
SCREEN 1    ' Auflösung 320 x 200 Bildpunkte
' Die Höhe dieses Rechtecks beträgt 100 Bildpunkte,
' und die Breite beträgt 120 Bildpunkte, wodurch das
' Verhältnis von Höhe zu Breite gleich 100/120 bzw.
' 5/6 ist. Das Ergebnis ist ein Quadrat:
LINE (50,50) - STEP (120,100),,B
```

Es folgt die Formel zur Berechnung des Seitenverhältnisses für einen gegebenen Bildschirmmodus:

(**4/3**) x (*yBildpunkte* / *xBildpunkte*)

wobei *xBildpunkte* x *yBildpunkte* der aktuellen Bildschirmauflösung entspricht. Im Bildschirmmodus 1 gibt diese Formel zum Beispiel das Seitenverhältnis (4/3) x (200/320), oder 5/6 an; im Bildschirmmodus 2 ist das Seitenverhältnis (4/3) x (200/640) oder 5/12.

Wenn Sie einen Bildschirm haben, dessen Verhältnis Breite zu Höhe nicht gleich 4:3 ist, gibt die folgende Formel eine allgemeinere Form zur Berechnung des Seitenverhältnisses an:

(*Bildschirmbreite* / *Bildschirmhöhe*) x (*yBildpunkte* / *xBildpunkte*)

Die **CIRCLE**-Anweisung kann zum Zeichnen von Ellipsen veranlaßt werden, indem die Werte des Argumentes *Aspekt* verändert werden, wie in Abschnitt 5.4.2 gezeigt.

## 5.5 Wie Sie ein grafisches Darstellungsfeld definieren

Die bisher dargestellten Grafikbeispiele verwenden alle den gesamten Bildschirm als Zeichenfläche mit absoluten Koordinatenabständen, die ab der äußersten oberen linken Ecke des Bildschirms gemessen werden.

Wenn Sie jedoch die Anweisung **VIEW** verwenden, können Sie auch eine Art Miniaturbildschirm (als "grafisches Darstellungsfeld" bezeichnet) innerhalb der Grenzen des physikalischen Bildschirms definieren. Ist ein grafisches Darstellungsfeld einmal definiert, finden alle grafischen Operationen innerhalb dieses Ausschnittes statt. Jede grafische Ausgabe außerhalb der Grenzen des Darstellungsfeldes wird "abgeschnitten"; das heißt, jeder Versuch, einen Punkt außerhalb des Darstellungsfeldes zu zeichnen, wird ignoriert.

Für die Verwendung eines Darstellungsfeldes gibt es zwei hauptsächliche Vorteile:

1. Ein Darstellungsfeld erleichtert es Ihnen die Größe und Position des Bildschirmbereiches, in dem Grafiken erscheinen, zu verändern.
2. Mit **CLS 1** können Sie den Bildschirm innerhalb eines Darstellungsfeldes löschen, ohne den Bildschirm außerhalb des Darstellungsfeldes zu zerstören.

***Hinweis*** Um zu lernen, wie ein "Text-Darstellungsfeld" für Ausgaben, die auf den Bildschirm geschrieben werden, angelegt wird, schlagen Sie im Abschnitt 3.1.6 im Kapitel 3, "Datei- und Geräte-E/A" nach.

Die allgemeine Schreibweise für **VIEW** (beachten Sie, daß die Option **STEP** mit **VIEW** nicht erlaubt ist) lautet

**VIEW** [[**SCREEN**] (*x1*,*y1*)-(*x2*,*y2*)[,[*Farbe*],[*Rand*]]]

wobei (*x1*,*y1*) und (*x2*,*y2*) mit Verwendung der BASIC-Standardschreibweise für Rechtecke (weitere Informationen finden Sie in Abschnitt 5.3.2.2, "Wie Sie Rechtecke zeichnen") die Ecken des Darstellungsfeldes definieren. Die wahlweisen Argumente *Farbe* und *Rand* erlauben Ihnen die Wahl einer Farbe für das Innere bzw. die Ränder des Darstellungsfeld-Rechteckes. Weitere Informationen zum Setzen und Verändern von Farben finden Sie in Abschnitt 5.7.

Die **VIEW**-Anweisung ohne Argumente läßt den gesamten Bildschirm zum Darstellungsfeld werden.

Die **VIEW**-Anweisung ohne die Option **SCREEN** läßt alle Bildpunktkoordinaten relativ zum Darstellungsfeld werden und nicht zum gesamten Bildschirm. Mit anderen Worten ist der nach der Anweisung

```
VIEW (50,60) - (150,175)
```

mit

```
PSET (10,10)
```

gezeichnete Bildpunkt sichtbar, da er sich 10 Bildpunkte unter und 10 Bildpunkte rechts der oberen linken Ecke des Darstellungsfeldes befindet. Beachten Sie, daß die Bildpunkte die absoluten Bildschirmkoordinaten (50+10,60+10) oder (60,70) damit erhalten.

Im Gegensatz dazu beläßt die **VIEW**-Anweisung ohne die **SCREEN**-Option alle Koordinaten absolut; das heißt, Koordinaten messen Abstände von der Seite des Bildschirms, nicht von den Seiten des Darstellungsfeldes. Daher ist der nach der Anweisung

```
VIEW SCREEN (50,60) - (150,175)
```

mit

```
PSET (10,10)
```

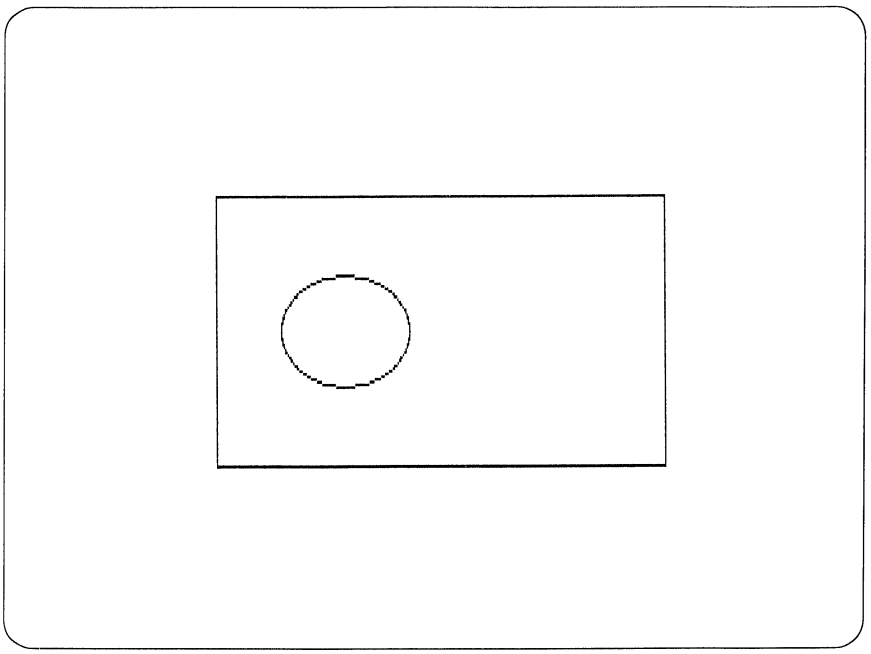
gezeichnete Bildpunkt nicht sichtbar, da er sich 10 Bildpunkte unter und 10 Bildpunkte rechts der oberen linken Ecke des Bildschirms und somit außerhalb des Darstellungsfeldes befindet.

### Beispiele

Die Ausgaben der nächsten beiden Beispiele sollten den Unterschied zwischen **VIEW** und **VIEW SCREEN** weiter verdeutlichen:

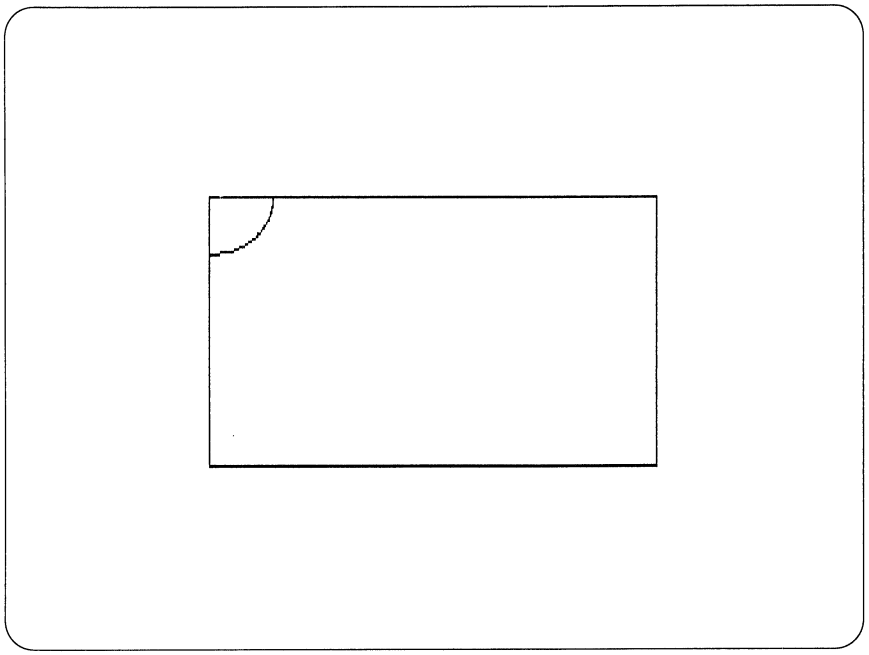
```
SCREEN 2
VIEW (100,50) - (450,150),,1
' Dieser Kreismittelpunkt hat die absoluten
' Koordinaten (100+100,50+50) bzw. (200,100):
CIRCLE (100,50), 50
```

Ausgabe mit **VIEW**



```
SCREEN 2
' Dieser Kreismittelpunkt hat die absoluten
' Koordinaten (100,50), so daß nur ein Teil des
' Kreises innerhalb des Darstellungsfeldes erscheint:
VIEW SCREEN (100,50) - (450,150), , 1
CIRCLE (100,50), 50
```

Ausgabe mit **VIEW SCREEN**



Beachten Sie, daß grafische Ausgabe, die sich außerhalb des aktuellen Darstellungsfeldes befindet, von den Feldseiten abgeschnitten wird und nicht auf dem Bildschirm erscheint.

## 5.6 Wie Sie die Koordinaten eines Darstellungsfeldes mit WINDOW neu definieren

Dieser Abschnitt zeigt Ihnen, wie Sie die Anweisung **WINDOW** und Ihr eigenes Koordinatensystem verwenden, um Bildschirmkoordinaten neu zu definieren.

In den Abschnitten 5.2 bis 5.5 stellen die zur Positionierung der Bildschirmpunkte auf dem Bildschirm verwendeten Koordinaten physikalische Abstände von der oberen linken Ecke des Bildschirms (oder der oberen linken Ecke des aktuellen Darstellungsfeldes, wenn ein solches mit einer **VIEW**-Anweisung definiert wurde) dar und werden als "physikalische Koordinaten" bezeichnet. Der "Ursprung", oder Referenzpunkt, für physikalische Koordinaten ist immer die obere linke Ecke des Bildschirms oder des Darstellungsfeldes und hat die Koordinaten (0,0). Wenn Sie sich auf dem Bildschirm nach unten und nach rechts bewegen, werden die x-Werte (horizontale Koordinaten) und y-Werte (vertikale Koordinaten) größer, wie in dem oberen Diagramm von Abbildung 5.5 gezeigt. Während dieses System für Bildschirme Standard ist, mag es Ihnen ungewohnt erscheinen, wenn Sie zum Zeichnen von Kurven andere Koordinaten verwendet haben. Zum Beispiel werden y-Werte einer Kurve in dem in der Mathematik verwendeten Kartesischen Koordinatensystem nach oben hin größer und nach unten hin kleiner.

Mit der BASIC-Anweisung **WINDOW** können Sie die Art, in der Bildpunkte adressiert werden, verändern. Damit können Sie jedes von Ihnen gewählte Koordinatensystem verwenden, wodurch Sie von der Einschränkung, physikalische Koordinaten benutzen zu müssen, befreit sind.

Die allgemeine Syntax für **WINDOW** ist

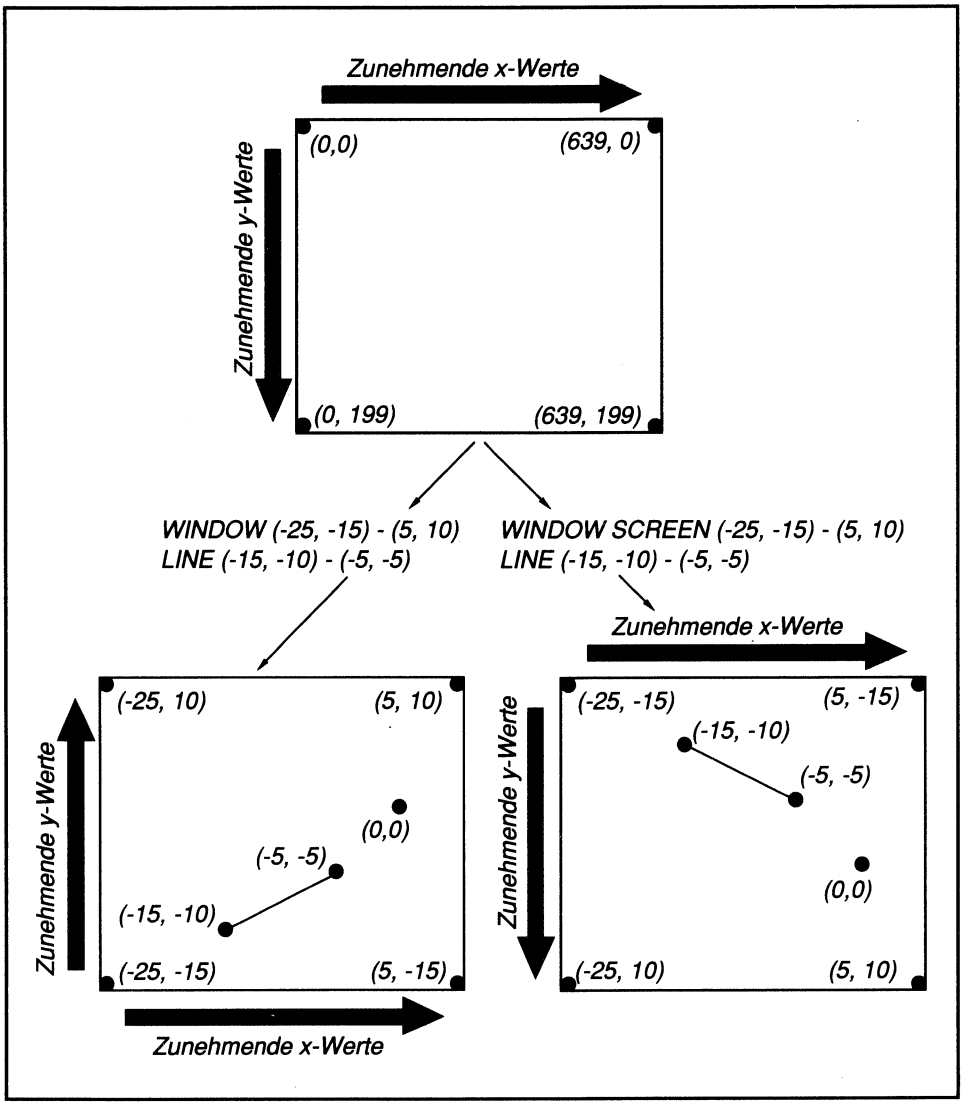
**WINDOW** [[**SCREEN**] (*x1*,*y1*)-(*x2*,*y2*)]

wobei *y1*, *y2*, *x1* und *x2* reelle Zahlen sind, die die obere, untere, linke und rechte Seite des Fensters angeben. Diese Zahlen werden als "logische Koordinaten" bezeichnet. Zum Beispiel teilt die folgende Anweisung Ihren Bildschirm neu ein, so daß er oben und unten von den Geraden `y = 10` bzw. `y = -15`, und links und rechts von den Geraden `x = -25` bzw.`x = 5` begrenzt ist:

```
WINDOW (-25,-15) - (5,10)
```

Nach einer **WINDOW**-Anweisung werden y-Werte zum oberen Ende des Bildschirms hin größer. Im Gegensatz dazu werden y-Werte nach einer Anweisung **WINDOW SCREEN** zum unteren Ende des Bildschirms hin größer. Die untere Hälfte von Abbildung 5.4 zeigt die Auswirkungen sowohl einer **WINDOW**- als auch einer **WINDOW SCREEN**-Anweisung auf eine im Bildschirmmodus 2 gezeichnete Gerade. Beachten Sie auch, wie diese beiden Anweisungen die Koordinaten der Bildschirmecken verändern.

*Abbildung 5.4* ***WINDOW*** *verglichen mit* ***WINDOW SCREEN***




***Hinweis*** Eine WINDOW-Anweisung ohne Argumente stellt das reguläre physikalische Koordinatensystem wieder her.

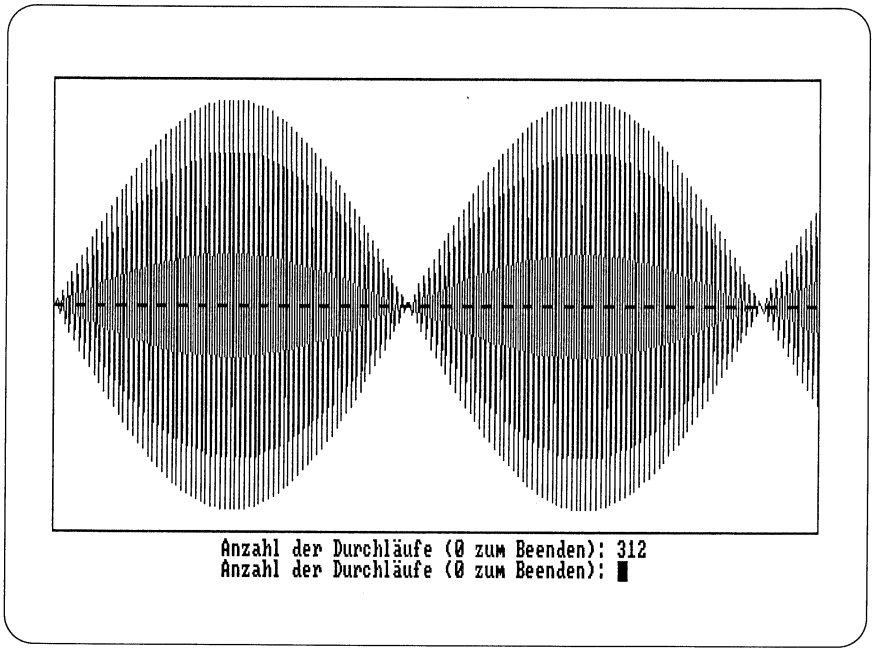
### Beispiel

Das folgende Beispiel verwendet sowohl **VIEW** als auch **WINDOW**, um das Schreiben eines Programmes zu vereinfachen, das die Sinus-Funktion für Winkelwerte von 0 Bogenmaß bis π Bogenmaß (oder 0° bis 180°) zeichnet.

Dieses Programm finden Sie auf Ihren Originaldisketten in der **SINUS.BAS** benannten Datei.

```
SCREEN 2
' Die Größe des Darstellungsfelds wird passend
' zur zu zeichnenden Kurve gewählt:
VIEW (20, 2)-(620, 172), , 1
CONST PI = 3.141592653589#
' Mache Fenster genügend groß, um Sinuskurve von
' 0 Bogenmaß bis ª Bogenmaß zu zeichnen:
WINDOW (0, -1.1)-(PI, 1.1)
Struktur% = &HFF00      ' Für unterbrochene Gerade.
VIEW PRINT 23 TO 24     ' Rolle geschriebene Ausgabe
                        ' in Zeile 23 und 24.
DO
   PRINT TAB(20);
   PRINT "Anzahl der Durchläufe ";
   INPUT "(0 zum Beenden): ", Durchlauf
   CLS
   LINE (PI, 0)-(0, 0),,,Struktur% 'Zeichne die (hori-
                                   'zontale) x-Achse.
   IF Durchlauf > 0 THEN
      ' Beginn bei (0,0) und zeichne die Kurve:
      FOR X = 0 TO PI STEP .01
          Y = SIN(Durchlauf * X)    ' Berechne die y-
                                    ' Koordinate.
          LINE -(X, Y)  ' Zeichne eine Gerade vom
                        ' letzten Punkt zu dem neuen
                        ' Punkt.
      NEXT X
   END IF
LOOP WHILE Durchlauf > 0
```

***Ausgabe***




## 5.6.1 Die Reihenfolge von Koordinatenpaaren

Genauso wie bei den anderen Grafikanweisungen von BASIC, die rechteckige Bereiche definieren (**GET**, **LINE** und **VIEW**), spielt in einer **WINDOW**-Anweisung die Reihenfolge der Koordinatenpaare keine Rolle. Daher hat das erste der unteren Anweisungspaare dieselbe Auswirkung wie das zweite Anweisungspaar:

```
VIEW (100,20) - (300,120)
WINDOW (-4,-3) - (0,0)
VIEW (300,120) - (100,20)
WINDOW (0,0) - (-4,-3)
```

## 5.6.2 Wie Sie sich logische und physikalische Koordinaten merken

Die Funktionen **PMAP** und **POINT** sind hilfreich, um sich physikalische und logische Koordinaten zu merken. **POINT**(*Zahl*) gibt Ihnen die aktuelle Position des grafischen Cursors an, indem es entweder die physikalischen oder logischen Koordinaten (abhängig vom Wert für *Zahl*) des letzten Punktes zurückgibt, auf den in einer Grafikanweisung Bezug genommen wurde. **PMAP** ermöglicht es Ihnen, physikalische in logische Koordinaten zu übertragen und umgekehrt.

***Hinweis*** Die von **PMAP** angegebenen physikalischen Koordinaten sind immer relativ zu dem aktuellen Darstellungsfeld.

### Beispiele

Das folgende Beispiel zeigt die unterschiedlichen Werte, die von **POINT**(*Zahl*) für *Zahl*-Werte von 0, 1, 2 oder 3 angegeben werden:

```
SCREEN 2
' Definiere das Fenster für logische Koordinaten:
WINDOW (-10,-30) - (-5,-10)
' Zeichne eine Gerade von dem Punkt mit den logischen
' Koordinaten (-9,-28) zu dem Punkt mit den
' logischen Koordinaten (-6,-24):
LINE (-9,-28) - (-6,-24)
PRINT "Physikalische x-Koordinate des letzten ";
PRINT "Punktes = " POINT(0)
PRINT "Physikalische y-Koordinate des letzten ";
PRINT "Punktes = " POINT(1)
PRINT
PRINT "Logische x-Koordinate des letzten ";
PRINT "Punktes = " POINT(2)
PRINT "Logische y-Koordinate des letzten ";
PRINT "Punktes = " POINT(3)
END
```

### *Ausgabe*

```
Physikalische x-Koordinate des letzten Punktes = 511
Physikalische y-Koordinate des letzten Punktes = 139
Logische x-Koordinate des letzten Punktes = -6
Logische y-Koordinate des letzten Punktes = -24
```

Mit der im vorhergehenden Beispiel angegebenen **WINDOW**-Anweisung würden die nächsten vier **PMAP**-Anweisungen die Ausgabe wie folgt schreiben:

```
' Bilde die logische x-Koordinate -6 auf die
' physikalische x-Koordinate ab und schreibe:
PhysX% = PMAP(-6,0)
PRINT PhysX%
' Bilde die logische y-Koordinate -24 auf die
' physikalische y-Koordinate ab und schreibe:
PhysY% = PMAP(-24,1)
PRINT PhysY%
' Bilde physikalisches x wieder auf logisches x ab und
' schreibe:
LogischX% = PMAP (PhysX%,2)
PRINT LogischX%
' Bilde physikalisches y wieder auf logisches y ab und
' schreibe:
LogischY% = PMAP (PhysY%,3)
PRINT LogischY%
```

***Ausgabe***

```
 511
 139
-6
-24
```

## 5.7 Wie Sie Farben verwenden

Wenn Sie eine CGA verwenden, können Sie nur zwischen den beiden folgenden Grafikmodi wählen:

1. **SCREEN 2** hat ein hohe Auflösung von 640 x 200 Bildpunkten mit nur einer Vordergrund- und nur einer Hintergrundfarbe. Dies wird als "monochrom" bezeichnet, da alle grafischen Ausgaben dieselbe Farbe haben.
2. **SCREEN 1** hat eine Auflösung von 320 x 200 Bildpunkten mit 4 Vordergrund- und 16 Hintergrundfarben.

Somit besteht eine Relation zwischen Farbe und Schärfe in den beiden Bildschirmmodi, die von der meisten Anzeige-/Adapter-Hardware unterstützt werden. Abhängig von den grafischen Fähigkeiten Ihres Systems müssen Sie die Schärfe nicht opfern, um den vollen Farbbereich zu erhalten (weitere Informationen zu Farbe in Bildschirmmodi höherer Auflösung finden Sie in der Beschreibung der **SCREEN**-Anweisung im *BASIC-Befehlsverzeichnis*).

Dieser Abschnitt beschäftigt sich ausschließlich mit den Bildschirmmodi 1 und 2.

## 5.7.1 Wie Sie eine Farbe für grafische Ausgabe wählen

Die folgende Liste zeigt, wohin Sie das Argument *Farbe* in den grafischen Anweisungen, die in vorhergehenden Abschnitten dieses Kapitels erläutert wurden, schreiben müssen. Diese Liste zeigt ebenso weitere Optionen (wie z. B. **BF** mit der Anweisung **LINE** oder *Rand* mit der Anweisung **VIEW**), die unterschiedliche Farben haben können. Beachten Sie bitte, daß für diese Anweisungen nicht die komplette Schreibweise angegeben ist. Diese Zusammenfassung hat allein die Absicht Ihnen zu zeigen, wie Sie die Option *Farbe* in solchen Anweisungen, die diese Option akzeptieren, einsetzen.

**PSET** (*x,y*), *Farbe*
**PRESET** (*x,y*), *Farbe*
**LINE** (*x1,y1*) - (*x2,y2*), *Farbe*[, **B** [**F**]]
**CIRCLE** (*x,y*), *Radius*, *Farbe*
**VIEW** (*x1,y1*) - (*x2,y2*), *Farbe*, *Rand*

Im Bildschirmmodus 1 ist das Argument *Farbe* ein numerischer Ausdruck mit den Werten 0, 1, 2 oder 3. Jeder dieser Werte, die als "Attribute" bezeichnet werden, stellt eine andere Farbe dar, wie im folgenden Beispiel gezeigt:

```
' Zeichne eine "unsichtbare" Gerade (in der Farbe des
' Hintergrundes):
LINE (10,10) - (310,10), 0
' Zeichne eine hellblaue (kobaltblaue) Gerade:
LINE (10,30) - (310,30), 1
' Zeichne eine violette Gerade:
LINE (10,50) - (310,50), 2
' Zeichne eine weiße Gerade:
LINE (10,70) - (310,70), 3
END
```

Wie in den Kommentaren des vorhergehenden Beispiels angemerkt, ergibt ein Wert 0 für *Farbe* keine sichtbare Ausgabe, da sie immer gleich der aktuellen Hintergrundfarbe ist. Zunächst scheint dies kein sinnvoller Farbwert zu sein, dennoch ist er hilfreich zum Entfernen von Abbildungen, ohne den gesamten Bildschirm oder das gesamte Darstellungsfeld zu löschen, wie im nächsten Beispiel gezeigt:

```
SCREEN 1
CIRCLE (100,100),80,2,,,3      ' Zeichne eine Ellipse.
Pause$ = INPUT$(1)             ' Warte auf eine
                               ' Tastenbetätigung.
CIRCLE (100,100),80,0,,,3      ' Entferne die Ellipse.
```

## 5.7.2 Wie Sie die Vordergrund- oder Hintergrundfarbe verändern

In Abschnitt 5.7.1 wurde bereits erwähnt, daß Sie 4 unterschiedliche Farben für grafische Ausgabe verwenden können. Diese 4 Farben beeinflussen den Vordergrund. Sie haben eine größere Vielfalt von Farben für die Hintergrundfarbe des Bildschirms zur Verfügung: 16 insgesamt.

Zusätzlich können Sie die Vordergrundfarbe mit der Verwendung einer unterschiedlichen "Palette" unmittelbar verändern. Im Bildschirmmodus 1 gibt es zwei Paletten oder Gruppen mit vier Farben. Jede Palette weist demselben Attribut eine unterschiedliche Farbe zu; zum Beispiel ist in Palette 1 (die Standardeinstellung) die mit dem Attribut 2 verknüpfte Farbe violett, während in Palette 0 die mit dem Attribut 2 verknüpfte Farbe rot ist. Wenn Sie eine CGA haben, sind diese Farben für jede Palette festgelegt; das heißt, die in Palette 1 der Nummer 2 zugewiesene Farbe ist immer violett, während die in Palette 0 mit Nummer 2 verknüpfte Farbe immer rot ist.

Wenn Sie eine EGA oder VGA haben, können Sie die Anweisung **PALETTE** verwenden, um die Farbe zu wählen, die von jedem Attribut angezeigt wird. Durch Veränderung der Argumente in der Anweisung **PALETTE** könnten Sie zum Beispiel die mit dem Attribut 1 angezeigte Farbe einmal grün und das nächste mal braun werden lassen. (Weitere Informationen zur Zuweisung von Farben finden Sie in Abschnitt 5.7.3, "Wie Sie Farben mit PALETTE und PALETTE USING verändern" in diesem Handbuch und in der Beschreibung der Anweisung **PALETTE** im *BASIC-Befehlsverzeichnis.*)

Im Bildschirmmodus 1 erlaubt Ihnen die **COLOR**-Anweisung die Steuerung sowohl der Hintergrundfarbe als auch der Palette der Vordergrundfarben. Es folgt die Schreibweise für **COLOR** im Bildschirmmodus 1:

**COLOR** [*Hintergrund*] [, *Palette*]

Das Argument *Hintergrund* ist ein numerischer Ausdruck von 0-15, und *Palette* ist ein numerischer Ausdruck, der entweder 0 oder 1 sein kann.

Die folgende Tabelle zeigt die Farben, die durch die 4 verschiedenen Vordergrundnummern in jeder der zwei Paletten erzeugt werden. Die folgende Tabelle zeigt die Farben, die mit den 16 verschiedenen Hintergrundnummern erzeugt werden.

| ***Nummer der Vordergrundfarbe*** | ***Farbe in Palette 0*** | ***Farbe in Palette 1*** |
|---|---|---|
| 0 | Aktuelle Hintergrundfarbe | Aktuelle Hintergrundfarbe |
| 1 | grün | kobaltblau (blau-grün) |
| 2 | rot | violett (hellpurpur) |
| 3 | braun | weiß (auf manchen Bildschirmen hellgrau) |

Die folgende Tabelle zeigt die Farben, die mit den 16 verschiedenen Hintergrundnummern erzeugt werden.

| ***Nummer der Hintergrundfarbe*** | ***Farbe*** |
|---|---|
| 0 | schwarz |
| 1 | blau |
| 2 | grün |
| 3 | kobaltblau |
| 4 | rot |
| 5 | violett |
| 6 | braun (auf manchen Bildschirmen dunkelgelb) |
| 7 | weiß (auf manchen Bildschirmen hellgrau) |
| 8 | dunkelgrau (auf manchen Bildschirmen schwarz) |
| 9 | hellblau |
| 10 | hellgrün |
| 11 | hellkobaltblau |
| 12 | hellrot |
| 13 | hellviolett |
| 14 | hellgelb (auf manchen Bildschirmen eventuell mit einem grünlichen Ton) |
| 15 | Leuchtend weiß oder sehr helles Grau |

Wie bereits in den Tabellen erwähnt, hängt der genaue Farbton und die Intensität verschiedener Farben von Ihrem Monitor ab.

### Beispiel

Das folgende Programm zeigt alle Kombinationen der 2 Farbpaletten mit den 16 unterschiedlichen Hintergrund-Bildschirmfarben.

Dieses Programm befindet sich in der Datei namens **FARBEN.BAS** auf den QuickBASIC-Originaldisketten.

```
SCREEN 1
Esc$ = CHR$(27)
' Zeichne drei Rechtecke und male das Innere jedes
' Rechtecks mit einer anderen Farbe aus:
FOR FarbWert = 1 TO 3
   LINE (X, Y)-STEP(60, 50), FarbWert, BF
   X = X + 61
   Y = Y + 51
NEXT FarbWert
LOCATE 21, 1
PRINT "ESC zum Beenden."
PRINT "Weiter mit jeder Taste."
' Begrenze zusätzlich geschriebene Ausgabe auf die
' 23. Zeile:
VIEW PRINT 23 TO 23
DO
   PaletteWert = 1
   DO
      ' PaletteWert ist entweder 1 oder 0:
      PaletteWert = 1 - PaletteWert
      ' Setze die Hintergrundfarbe und wähle die
      ' Palette:
      COLOR HintGrundWert, PaletteWert
      PRINT "Hintergrund ="; HintGrundWert;
      PRINT "Palette ="; PaletteWert;
      Pause$ = INPUT$(1)   ' Warte auf eine
                           ' Tastenbetätigung.
      PRINT
   ' Verlasse die Schleife, wenn beide Paletten
   ' gezeigt wurden, oder wenn der Benutzer die ESC-
   ' Taste betätigt hat:
   LOOP UNTIL PaletteWert = 1 OR Pause$ = Esc$
   HintGrundWert = HintGrundWert + 1
```

```
' Verlasse diese Schleife, wenn alle 16
' Hintergrundfarben gezeigt wurden, oder wenn der
' Benutzer die ESC-Taste betätigt hat:
LOOP UNTIL HintGrundWert > 15 OR Pause$ = Esc$

SCREEN 0          ' Stelle den Textmodus und 80-Spalten
WIDTH 80          ' Bildschirmbreite wieder her.
```

## 5.7.3 Wie Sie Farben mit PALETTE und PALETTE USING verändern

***Hinweis*** Wenn Sie keine EGA oder VGA besitzen, können Sie diesen Abschnitt überschlagen.

Der vorhergehende Abschnitt zeigte, wie Sie die von einem Attribut angezeigte Farbe einfach durch die Festlegung einer anderen Palette in der **COLOR**-Anweisung verändern können. Sie sind damit jedoch auf zwei feste Farbpaletten mit nur je vier Farben eingeschränkt. Darüberhinaus kann jedes Attribut nur für eine von zwei möglichen Farben stehen; das Attribut 1 kann zum Beispiel nur grün oder kobaltblau festlegen.

Die Anweisungen **PALETTE** bzw. **PALETTE USING** geben Ihnen dagegen eine erheblich größere Flexibilität, die Farbpalette zu verändern. Bei Verwendung dieser Anweisungen können Sie jede Farbe der verfügbaren Palette jedem Attribut zuweisen. Zum Beispiel erscheint nach der folgenden Anweisung die Ausgabe aller Grafikanweisungen, die Attribut 4 verwenden, in hellviolett (Farbe 13):

```
PALETTE 4, 13
```

Diese Farbveränderung ist unverzüglich und betrifft nicht nur nachfolgende Grafikanweisungen, sondern auch jede bereits auf dem Bildschirm befindliche Ausgabe. Mit anderen Worten können Sie Ihren Bildschirm zeichnen und malen und dann die Palette umschalten, um einen sofortigen Farbwechsel durchzuführen, wie im folgenden Beispiel gezeigt:

```
SCREEN 8
LINE (50,50) - (150,150), 4    ' Zeichne eine Gerade in
                               ' rot.

PALETTE 4, 13      ' Attribut 4 entspricht jetzt Farbe
                   ' 13, so daß die in der letzten
                   ' Anweisung gezeichnete Gerade jetzt
                   ' hellviolett ist.
```

Mit der Option **USING** der Anweisung **PALETTE** können Sie die den Attributen zugewiesenen Farben alle auf einmal ändern.

### Beispiel

Das folgende Beispiel verwendet die Anweisung **PALETTE USING**, um den Eindruck von Bewegung auf dem Bildschirm zu vermitteln, indem die mit den Attributen 1 bis 15 angezeigten Farben ständig umlaufen.

Dieses Programm befindet sich in der **PALETTE.BAS** benannten Datei auf Ihren QuickBASIC-Originaldisketten.

```
DECLARE SUB InitPalette ()
DECLARE SUB WechselPalette ()
DECLARE SUB ZeichneEllipsen ()

DEFINT A-Z
DIM SHARED PaletteFeld(15)
SCREEN 8              ' Auflösung 640 x 200; 16 Farben

InitPalette           ' Initialisiere PaletteFeld.
ZeichneEllipsen       ' Zeichne und male konzentrische
                      ' Ellipsen.

DO                    ' Schalte die Palette um, bis eine
                      ' Taste betätigt wird.
   WechselPalette
LOOP WHILE INKEY$ = ""

END
'
' ================== InitPalette ====================
'    Diese Prozedur initialisiert das zum Wechseln der
'    Palette verwendete ganzzahlige Datenfeld.
' ===================================================
'
SUB InitPalette STATIC
   FOR I = 0 TO 15
      PaletteFeld(I) = I
   NEXT I
END SUB
'
' ================ ZeichneEllipsen ==================
'    Diese Prozedur zeichnet 15 konzentrische Ellipsen
'    und malt das Innere einer jeden mit einer anderen
'    Farbe aus.
' ===================================================
'
SUB ZeichneEllipsen STATIC
   CONST ASPEKT = 1 / 3
```

```
   FOR FarbWert = 15 TO 1 STEP -1
      Radius = 20 * FarbWert
      CIRCLE (320, 100), Radius, FarbWert, , , ASPEKT
      PAINT (320, 100), FarbWert
   NEXT
END SUB
'
' ================= WechselPalette ==================
'    Diese Prozedur läßt die Palette bei jedem Aufruf
'    um eins rotieren. Z. B. ist nach dem ersten
'    Aufruf von WechselPalette PaletteFeld(1) = 2,
'    PaletteFeld(2) = 3, . . . , PaletteFeld(14) = 15
'    und PaletteFeld(15) = 0
' ====================================================
'
SUB WechselPalette STATIC
   FOR I = 1 TO 15
      PaletteFeld(I) = (PaletteFeld(I) MOD 15) + 1
   NEXT I
   PALETTE USING PaletteFeld(0) ' Schalte die von
                                ' jeder der 15
                                ' Attribute angezeigte
                                ' Farbe um.
END SUB
```

## 5.8 Wie Sie Formen ausmalen

Abschnitt 5.3.2.2 zeigt Ihnen, wie ein Rechteck mit der Option **B** der **LINE**-Anweisung gezeichnet wird, und wie dann das Rechteck durch Hinzufügen der Option **F** (für Füllen) ausgemalt wird:

```
SCREEN 1
' Zeichne ein Quadrat, male das Innere dann mit Farbe
' 1 aus (Kobaltblau in der Standard-Palette):
LINE (50,50) - (110,100),1,BF
```

Mit der BASIC-Anweisung **PAINT** können Sie jede geschlossene Abbildung mit jeder von Ihnen gewählten Farbe füllen. **PAINT** erlaubt es Ihnen ebenso, geschlossene Abbildungen mit Ihren eigenen gewohnten Mustern wie Streifen und Karos auszufüllen, wie in Abschnitt 5.8.2, "Zeichnen mit Mustern" gezeigt.

## 5.8.1 Ausmalen mit Farben

Um eine geschlossene Form mit einer durchgehenden Farbe auszumalen, benutzen Sie diese Form der Anweisung **PAINT**:

**PAINT** [**STEP**] (*x*,*y*) [, [*Innen*], [*Rand*]]

Hier sind *x*,*y* die Koordinaten eines Punktes im Inneren der Abbildung, die Sie ausmalen wollen; *Innen* ist die Nummer der Farbe, mit der Sie malen wollen, und *Rand* die Farbnummer der Kontur der Abbildung.

Zum Beispiel zeichnen die folgenden Programmzeilen einen Kreis in kobaltblau und malen dann das Innere des Kreises violett aus:

```
SCREEN 1
CIRCLE (160,100), 50, 1
PAINT (160,100), 2,1
```

Für das Ausmalen von Abbildungen gelten die folgenden drei Regeln:

1. Die in der **PAINT**-Anweisung angegebenen Koordinaten müssen sich auf einen Punkt innerhalb der Abbildung beziehen.

   Zum Beispiel würde jede der folgenden Anweisungen dieselbe Auswirkung wie die in den zwei vorhergehenden Beispielen gezeigten **PAINT**-Anweisungen haben, da alle der Koordinaten Punkte innerhalb des Kreises kennzeichnen:

   ```
   PAINT (150,90), 2, 1
   PAINT (170,110), 2, 1
   PAINT (180,80), 2, 1
   ```

   Im Gegensatz dazu würde die nächste Anweisung, da `(5,5)` einen Punkt außerhalb des Kreises kennzeichnet, den gesamten Bildschirm mit Ausnahme des Kreisinneren, das die aktuelle Hintergrundfarbe behält, ausmalen:

   ```
   PAINT (5,5), 2, 1
   ```

Wenn die Koordinaten einer **PAINT**-Anweisung einen Punkt direkt auf dem Rand der Abbildung angeben, wird nicht ausgemalt:

```
LINE (50,50) - (150,150),,B   ' Zeichne ein Rechteck.
PAINT (50, 100)               ' Der Punkt mit den Koordinaten
                              ' (50,100) liegt auf dem oberen
                              ' Rand des Rechtecks, daher
                              ' wird nicht ausgemalt.
```

**2.** Die Abbildung muß ganz geschlossen sein; andernfalls "läuft" die Malfarbe "aus", wodurch der gesamte Bildschirm oder das gesamte Darstellungsfeld (oder jede größere Abbildung, die die erste ganz umschließt) gefüllt wird.

In dem folgenden Programm zeichnet die Anweisung **CIRCLE** zum Beispiel eine Ellipse, die nicht ganz komplett ist (auf der rechten Seite befindet sich eine kleine Öffnung); die Anweisung **LINE** umschließt dann die Teilellipse mit einem Rechteck. Obwohl das Ausmalen im Inneren der Ellipse beginnt, fließt die Malfarbe durch die Öffnung und füllt das gesamte Rechteck aus.

```
SCREEN 2
CONST PI = 3.141592653589#
CIRCLE (300,100), 80, , 0, 1.9*PI, 3
LINE (200,10) - (400,190), , B
PAINT (300,100)
```

**3.** Wenn Sie ein Objekt mit einer anderen Farbe als der zur Umrandung des Objektes verwendeten ausmalen, müssen Sie die Option *Rand* verwenden, um **PAINT** mitzuteilen, wo das Ausmalen beendet werden soll.

Zum Beispiel zeichnet das folgende Programm ein grün (Attribut 1 in Palette 0) umrandetes Dreieck und versucht anschließend, das Innere des Dreiecks rot (Attribut 2) auszumalen. Da die **PAINT**-Anweisung jedoch nicht anzeigt, wo das Ausmalen beendet werden soll, malt diese den gesamten Bildschirm rot aus.

```
SCREEN 1
COLOR ,0
LINE (10,25) - (310,25),1
LINE - (160,175),1
LINE - (10,25),1
PAINT (160,100), 2
```

Die folgende Änderung der **PAINT**-Anweisung (für das Innere rot wählen und beenden, wenn ein grünfarbiger Rand erreicht wird) erzielt den gewünschten Effekt:

```
PAINT (160,100), 2, 1
```

Beachten Sie, daß Sie in der **PAINT**-Anweisung keine Randfarbe angeben müssen, wenn die Farbe zum Ausmalen dieselbe ist wie die Randfarbe.

```
LINE (10,25) - (310,25),1
LINE - (160,175),1
LINE - (10,25),1
PAINT (160,100), 1
```

## 5.8.2 Zeichnen mit Mustern: Ausfüllen mit Mustern

Sie können die **PAINT**-Anweisung dazu verwenden, eine geschlossene Abbildung mit einem Muster zu füllen: Dieser Vorgang wird als "Ausfüllen mit Mustern" bezeichnet. Dies können Sie sich so ähnlich vorstellen wie das Erzeugen eines Musters auf dem Fußboden, das aus dutzenden separater aber identischer Kacheln gebildet wird.

Wenn Sie Ausfüllen mit Mustern verwenden, ist das Argument *Innen* in der Syntax von **PAINT** ein Zeichenkettenausdruck und keine Zahl. Da *Innen* jeder Zeichenkettenausdruck sein kann, besteht eine bequeme Möglichkeit zur Definition von "Kachelmustern" für *Innen* in der folgenden Form:

**CHR$**(*Arg1*) + **CHR$**(*Arg2*) + **CHR$**(*Arg3*) + ... + **CHR$**(*Argn*)

Hier sind *Arg1*, *Arg2* und so weiter 8-Bit-Ganzzahlen. Eine Erklärung, wie Sie diese 8-Bit-Ganzzahlen bilden, finden Sie in den Abschnitten 5.8.2.2 bis 5.8.2.4.

### 5.8.2.1 Größe eines Kachelmusters in unterschiedlichen Bildschirmmodi

Jede Kachel eines Musters ist aus einem rechteckigen Raster von Bildpunkten zusammengesetzt. Dieses Kachelraster kann in allen Bildschirmmodi bis zu 64 Zeilen haben. Die Anzahl an Bildpunkten in jeder Zeile hängt jedoch vom Bildschirmmodus ab.

Aus folgenden Gründen variiert die Länge jeder Kachelzeile abhängig vom Bildschirmmodus: Obwohl die Anzahl an Bits in jeder Zeile auf acht festgelegt ist (die Länge einer Ganzzahl), verkleinert sich die Anzahl an Bildpunkten, die diese acht Bits darstellen können, im gleichen Maße, wie sich die Anzahl an Farbattributen in einem gegebenen Bildschirmmodus vergrößert.

Zum Beispiel ist im Bildschirmmodus 2, der nur 1 Farbattribut hat, die Anzahl der Bits pro Bildpunkt gleich 1; im Bildschirmmodus 1, der 4 verschiedene Attribute hat, ist die Anzahl der Bits pro Bildpunkt gleich 2; im EGA-Bildschirmmodus 7, der 16 Attribute hat, ist die Anzahl der Bits pro Bildpunkt gleich 4. Die folgende Formel ermöglicht Ihnen die Berechnung der Bits pro Bildpunkt in jedem gegebenen Bildschirmmodus:

$$\textit{Bits pro Bildpunkt} = \mathrm{LOG}_2(\textit{AnzAttribute})$$

Hierbei ist *AnzAttribute* die Anzahl an Farbattributen in diesem Bildschirmmodus.

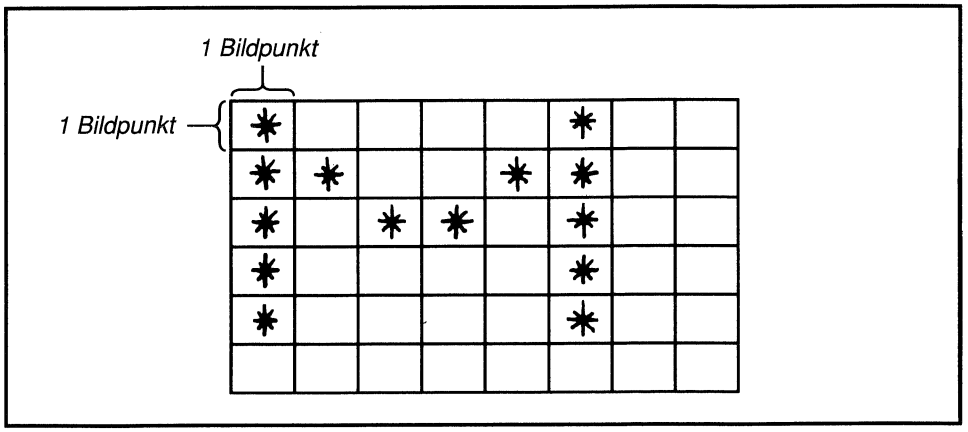
Daher ist die Länge einer Kachelzeile im Bildschirmmodus 2 acht Bildpunkte (acht Bits dividiert durch ein Bit pro Bildpunkt), aber nur vier Bildpunkte im Bildschirmmodus 1 (acht Bits dividiert durch zwei Bits pro Bildpunkt).

Die nächsten drei Abschnitte zeigen den schrittweisen Ablauf, der mit dem Erzeugen eines Kachelmusters verbunden ist. Abschnitt 5.8.2.2 zeigt, wie im Bildschirmmodus 2 ein monochromes Muster erzeugt wird. Der nächste Abschnitt 5.8.2.3 zeigt, wie im Bildschirmmodus 1 ein vielfarbiges Muster erzeugt wird. Wenn Sie eine EGA-Karte haben, können Sie in Abschnitt 5.8.2.4 nachlesen, wie im Bildschirmmodus 8 ein vielfarbiges Muster erzeugt wird.

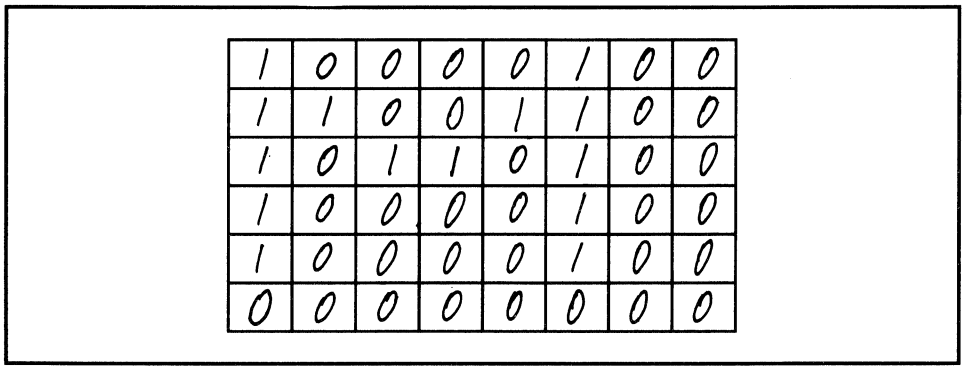
### 5.8.2.2 Wie Sie im Bildschirmmodus 2 ein einfarbiges Muster erzeugen

Die folgenden Schritte zeigen, wie ein Kachelmuster definiert und verwendet wird, das dem Buchstaben "M" gleicht:

1. Zeichnen Sie das Muster einer Kachel in einem Raster mit 8 Spalten und mit sovielen Zeilen, wie Sie benötigen (bis zu 64). In diesem Beispiel hat die Kachel 6 Zeilen; ein Stern (*) in einem Rechteck bedeutet, der Bildpunkt ist eingeschaltet:



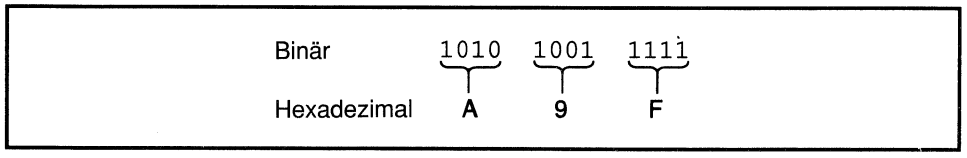
2. Übersetzen Sie nun jede Bildpunktzeile in eine 8-Bitzahl, wobei eine 1 bedeutet, der Bildpunkt ist eingeschaltet, und eine 0 bedeutet, der Bildpunkt ist ausgeschaltet:




3. Wandeln Sie die in Schritt 2 gegebenen binären Zahlen in hexadezimale Ganzzahlen um:

```
10000100 = &H84
11001100 = &HCC
10110100 = &HB4
10000100 = &H84
10000100 = &H84
00000000 = &H00
```

Diese Ganzzahlen müssen nicht hexadezimal sein; sie können auch dezimal oder oktal sein. Die meisten Leute finden jedoch die Umwandlung von binär in hexadezimal leichter. Um zum Beispiel von binär in hexadezimal umzuwandeln, lesen Sie die Binärzahlen von rechts nach links und fassen Sie jeweils vier binären Ziffern zu einer Gruppe zusammmen. Jede dieser Gruppen von vier Ziffern wird anschließend in die hexadezimale Entsprechung umgewandelt, wie im folgenden Diagramm gezeigt:




Die nachfolgende Tabelle führt 4-Bit-Binärsequenzen und deren hexadezimale Entsprechungen auf.

4. Erzeugen Sie eine Zeichenkette durch Verketten der Zeichen mit den ASCII-Werten aus Schritt 3 (verwenden Sie die Funktion **CHR$**, um diese Zeichen zu erhalten):

```
Kachel$ = CHR$(&H84) + CHR$(&HCC) + CHR$(&HB4) + _
          CHR$(&H84) + CHR$(&H84) + CHR$(&H00)
```

5. Zeichnen Sie eine Abbildung und malen Sie deren Inneres aus, indem Sie **PAINT** und das Zeichenkettenargument aus Schritt 4 verwenden:

```
PAINT (X,Y), Kachel$
```

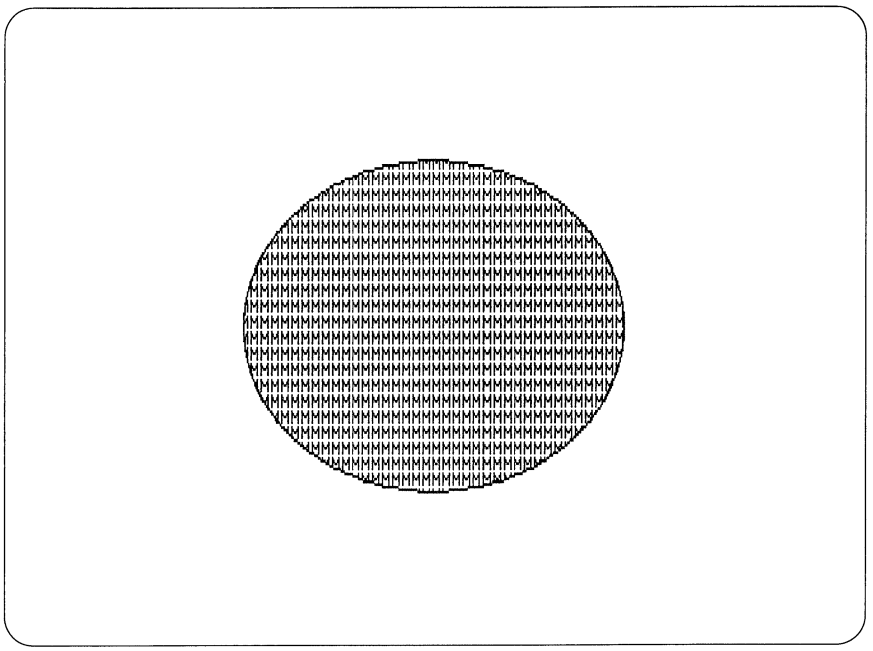
| *Binärzahl* | *Hexadezimalzahl* |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

### Beispiel

Das folgende Beispiel zeichnet einen Kreis und malt anschließend das Kreisinnere mit dem in den vorhergehenden Schritten erzeugten Muster aus.

```
SCREEN 2
CLS
Kachel$ = CHR$(&H84) + CHR$(&HCC) + CHR$(&HB4) + _
          CHR$(&H84) + CHR$(&H84) + CHR$(&H00)
CIRCLE STEP (0,0),150
PAINT STEP (0,0), Kachel$
```

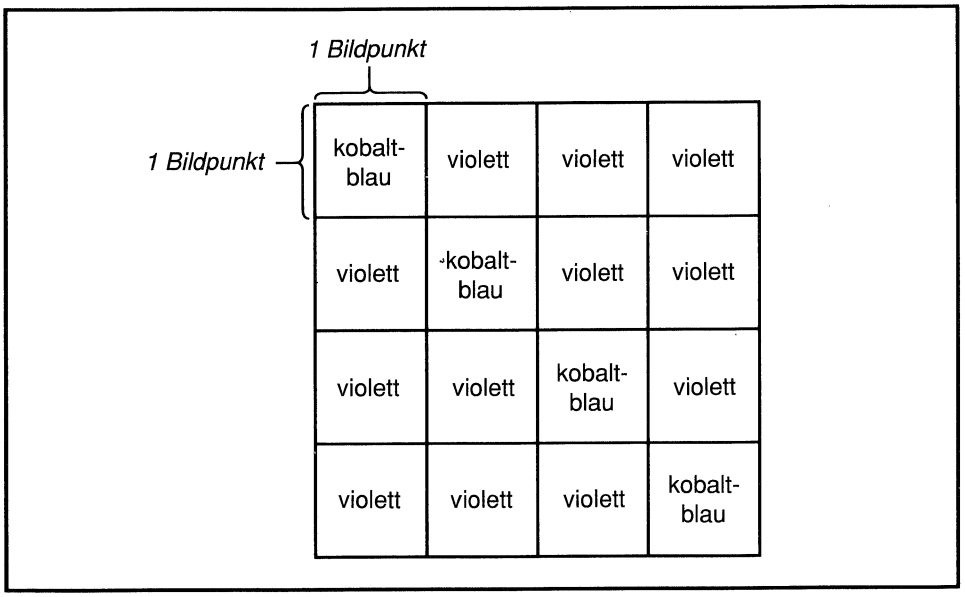
***Ausgabe***



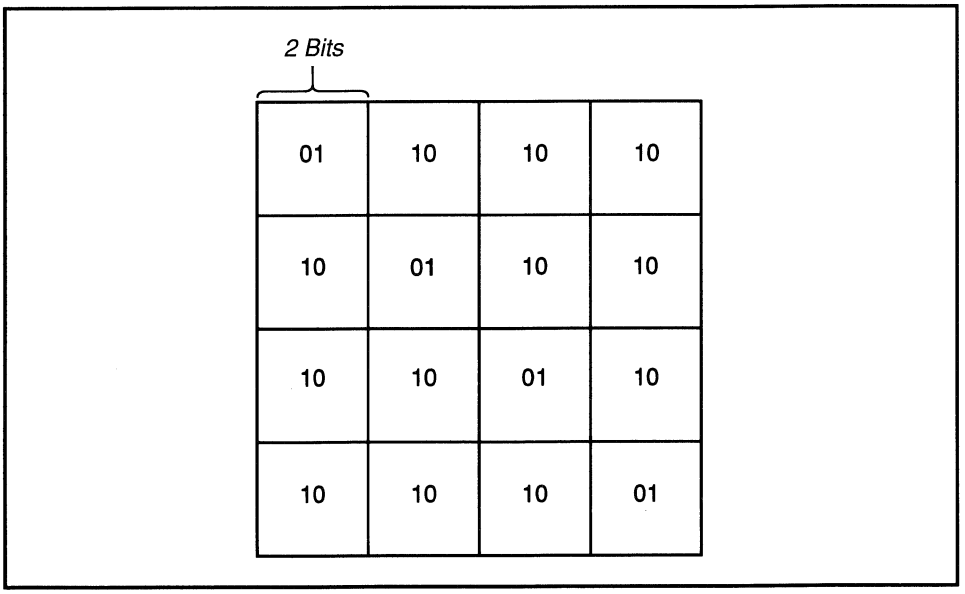
### 5.8.2.3 Wie Sie im Bildschirmmodus 1 ein vielfarbiges Muster erzeugen

Die folgenden Schritte zeigen, wie Sie ein vielfarbiges Muster erzeugen, das aus alternierenden kobaltblauen bzw. violetten diagonalen Streifen besteht (oder grün bzw. rot mit Palette 0):

1. Zeichnen Sie das Kachelmuster in einem Raster mit 4 Spalten (4 Spalten, da jede Bildpunktzeile in einer 8-Bit-Ganzzahl gespeichert ist und jeder Bildpunkt im Bildschirmmodus 1 zwei Bits erfordert) und sovielen Zeilen wie Sie benötigen (bis zu 64). In diesem Beispiel hat die Kachel 4 Zeilen, wie im nächsten Diagramm gezeigt.




**2.** Wandeln Sie die Farben in deren zugehörige Farbnummern in binärer Schreibweise um, wie unten gezeigt (stellen Sie sicher, daß Sie 2-Bitwerte verwenden, also 1 = binär 01 und 2 = binär 10):



3. Wandeln Sie die Binärzahlen aus Schritt 2 in hexadezimale Ganzzahlen um:

```
01101010 = &H6A
10011010 = &H9A
10100110 = &HA6
10101001 = &HA9
```

4. Erzeugen Sie eine Zeichenkette durch Verketten der Zeichen mit den ASCII-Werten aus Schritt 3 (verwenden Sie die Funktion **CHR$**, um diese Zeichen zu erhalten):

```
Kachel$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + _
          CHR$(&HA9)
```

5. Zeichnen Sie eine Abbildung und malen Sie deren Inneres aus, indem Sie **PAINT** und das Zeichenkettenargument aus Schritt 4 verwenden:

```
PAINT (X,Y), Kachel$
```

Das folgende Programm zeichnet ein Dreieck und malt anschließend dessen Inneres mit dem in den vorhergehenden Schritten erzeugten Muster aus.

```
SCREEN 1
' Definiere ein Muster:
Kachel$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + _
          CHR$(&HA9)
' Zeichne ein Dreieck in weiß (Farbe 3):
LINE (10,25) - (310,25)
LINE - (160,175)
LINE - (10,25)
' Male das Innere des Dreiecks mit dem Muster aus:
PAINT (160,100), Kachel$
```

Beachten Sie, daß wenn die Abbildung, die Sie ausmalen wollen, mit einer Farbe umrandet ist, die auch in dem Muster enthalten ist, Sie in **PAINT** das Argument *Rand*, wie nachfolgend gezeigt, angeben müssen; andernfalls geht das Muster über die Ränder der Abbildung hinaus.

```
SCREEN 1
' Definiere ein Muster:
Kachel$ = CHR$(&H6A) + CHR$(&H9A) + CHR$(&HA6) + _
          CHR$(&HA9)
' Zeichne ein Dreieck in violett (Farbe 2):
LINE (10,25) - (310,25),2
LINE - (160,175),2
LINE - (10,25),2
' Male das Innere des Dreiecks mit dem Muster aus und
' füge das Randargument (,2) hinzu, um PAINT
' mitzuteilen, wo mit dem Ausmalen aufgehört werden
' soll:
PAINT (160,100), Kachel$, 2
```

Machmal möchten Sie vielleicht, nachdem Sie eine Abbildung mit einer Farbe oder einem Muster ausgemalt haben, diese Abbildung oder einige ihrer Teile erneut ausmalen. Wenn das neue Muster zwei oder mehr angrenzende Zeilen enthält, die identisch mit dem aktuellen Hintergrund der Abbildung sind, werden Sie feststellen, daß das Ausmalen mit Mustern nicht funktioniert. Stattdessen breitet sich das Muster aus, umgeben von Bildpunkten, die dieselben wie zwei oder mehr seiner Zeilen sind, und das Ausmalen wird beendet.

Sie können dieses Problem abschwächen, indem Sie in **PAINT** das Argument *Hintergrund* verwenden, wenn sich nicht mehr als zwei angrenzende Zeilen in Ihrem neuen Muster befinden, die das gleiche Muster aufweisen wie der alte Hintergrund. **PAINT** mit *Hintergrund* hat die folgende Schreibweise:

**PAINT** [**STEP**] (*x*,*y*) [, [*Innen*] [, [*Rand*] [, *Hintergrund*]]]

Das Argument *Hintergrund* ist ein Zeichen der Form **CHR$**(*n*), das die Zeilen im Kachelmuster angibt, die dieselben Muster aufweisen wie die des aktuellen Abbildungshintergrundes. Kurz gesagt teilt *Hintergrund* **PAINT** mit, diese Zeilen beim erneuten Ausmalen der Abbildung zu überspringen. Das nächste Beispiel verdeutlicht, wie dies funktioniert.

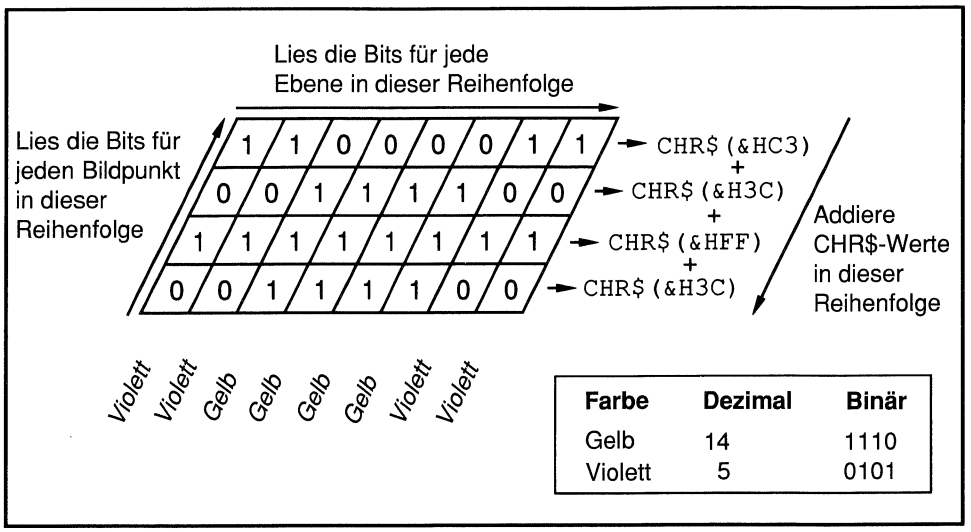
```
SCREEN 1
' Definiere ein Muster:
Kachel$ = CHR$(&H55) + CHR$(&H55) + _   ' 2 Zeilen in
                                        ' kobaltblau
          CHR$(&HAA) + CHR$(&HAA) + _   ' 2 Zeilen in
                                        ' violett
          CHR$(&HFF) + CHR$(&HFF)       ' 2 Zeilen in
                                        ' weiß
' Zeichne ein Dreieck in weiß (Farbe Nummer 3):
LINE (10,25) - (310,25)
LINE - (160,175)
LINE - (10,25)
' Male das Innere in violett aus:
PAINT (160,100),2,3
' Warte auf eine Tastenbetätigung:
Pause$ = INPUT$(1)
' Da der Hintergrund bereits violett ist, teilt
' CHR$(&HAA) PAINT mit, die violetten Zeilen des
' Kachelmusters zu überspringen:
PAINT (160,100), Kachel$,,CHR$(&HAA)
```

### 5.8.2.4 Wie Sie im Bildschirmmodus 8 ein vielfarbiges Muster erzeugen

In den Bildschirmmodi EGA bzw. VGA werden mehr als eine 8-Bit-Ganzzahl benötigt, um eine Zeile einer Musterkachel zu definieren. In diesen Bildschirmmodi besteht eine Zeile aus mehreren Ebenen von 8-Bit-Ganzzahlen. Der Grund dafür ist, daß ein Bildpunkt dreidimensional in einem Stapel von "Bitebenen" dargestellt wird, und nicht sequentiell in einer einzigen Ebene, wie es in den Bildschirmmodi 1 und 2 der Fall ist. Zum Beispiel hat der Bildschirmmodus 8 vier dieser Bitebenen. Daher befindet sich jedes der vier Bits pro Bildpunkt in diesem Bildschirmmodus auf einer unterschiedlichen Ebene. (Weitere Informationen zu der Anzahl von Ebenen in jedem Bildschirmmodus finden Sie in der Beschreibung zur Anweisung **SCREEN** im *BASIC-Befehlsverzeichnis*.)

Die folgenden Schritte veranschaulichen die Abfolge zur Erzeugung eines vielfarbigen Musters, das aus abwechselnd gelben und violetten Zeilen besteht. Beachten Sie, wie jede Zeile der Musterkachel mit vier parallelen Bytes dargestellt wird:

1. Definieren Sie eine Zeile von Bildpunkten in der Musterkachel. Jeder Bildpunkt in der Zeile beansprucht vier Bits und jedes Bit befindet sich auf einer anderen Ebene, wie nachfolgend gezeigt.

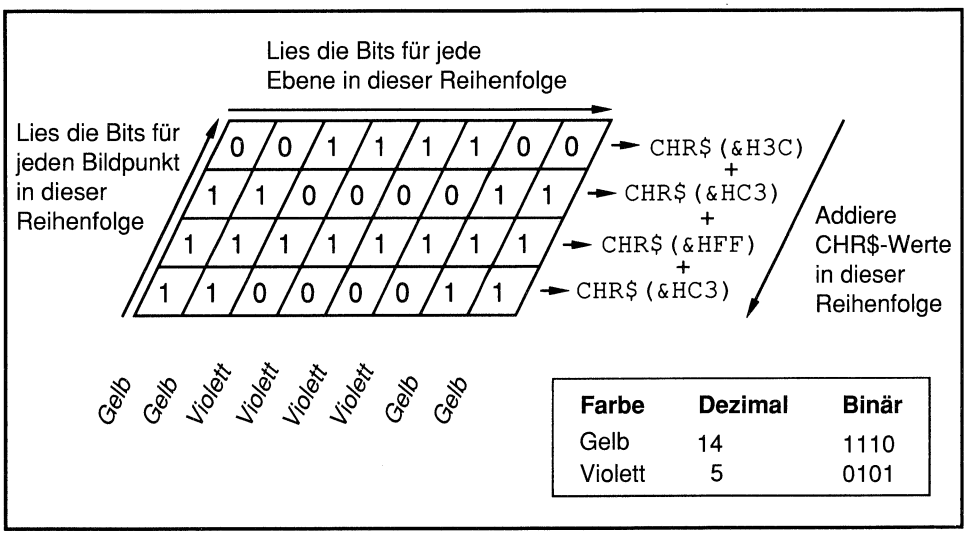



| Farbe | Dezimal | Binär |
|---|---|---|
| Gelb | 14 | 1110 |
| Violett | 5 | 0101 |

Addieren Sie die CHR$-Werte aller vier Bitebenen, um ein Kachelbyte zu bekommen. Diese Zeile wird in der Musterkachel wiederholt, also:

```
Zeile$(1) = Zeile$(2)= _
            CHR$(&HC3) + CHR$(&H3C) + CHR$(&HFF) + _
            CHR$(&H3C)
```

**2.** Definieren Sie eine weitere Zeile von Bildpunkten in der Musterkachel, wie folgt:




| Farbe | Dezimal | Binär |
|---|---|---|
| Gelb | 14 | 1110 |
| Violett | 5 | 0101 |

Diese Zeile wird ebenfalls in der Musterkachel wiederholt, also:

```
Zeile$(3) = Zeile$(4)= _
            CHR$(&H3C) + CHR$(&HC3) + CHR$(&HFF) + _
            CHR$(&HC3)
```

## Beispiel

Das folgende Beispiel zeichnet ein Rechteck und malt anschließend dessen Inneres mit dem in den vorhergehenden Schritten erzeugten Muster aus:

```
SCREEN 8
DIM Zeile$(1 TO 4)
' Zwei Zeilen mit abwechselnd violett und gelb:
Zeile$(1) = CHR$(&HC3) + CHR$(&H3C) + CHR$(&HFF) +_
            CHR$(&H3C)
Zeile$(2) = Zeile$(1)
' Drehe das Muster um (zwei Zeilen abwechselnd gelb
' und violett):
Zeile$(3) = CHR$(&H3C) + CHR$(&HC3) + CHR$(&HFF) + _
            CHR$(&HC3)
Zeile$(4) = Zeile$(3)
```

```
' Erzeuge eine Musterkachel mit den oben definierten
' Zeilen:
FOR I% = 1 TO 4
   Kachel$ = Kachel$ + Zeile$(I%)
NEXT I%
' Zeichne ein Rechteck und fülle es mit dem Muster
' aus:
LINE (50, 50)-(570, 150), , B
PAINT (320, 100), Kachel$
```

## 5.9 DRAW: Eine graphische Makro-Sprache

Die Anweisung **DRAW** ist selbst eine "Miniatursprache". Sie zeichnet und malt Bilder auf den Bildschirm, wobei sie als "Makros" bezeichnete Befehlssätze verwendet, die aus einem oder zwei Buchstaben bestehen und in einen Zeichenkettenausdruck eingefügt sind.

**DRAW** bietet die folgenden Vorteile gegenüber den anderen bisher erläuterten Grafikanweisungen:

- Das Makro-Zeichenkettenargument in **DRAW** ist kurz: Eine einzige kurze Zeichenkette kann dieselbe Ausgabe erzeugen, wie mehrere **LINE**-Anweisungen.
- Mit **DRAW** erzeugte Bilder können leicht maßstabgetreu verändert werden, d.h. verkleinert oder vergrößert werden, indem in der Makro-Zeichenkette das Makro **S** verwendet wird.
- Mit **DRAW** erzeugte Bilder können um eine beliebige Gradzahl gedreht werden, indem in der Makro-Zeichenkette das Makro **TA** verwendet wird.

Ein komplette Liste der mit **DRAW** verwendeten Makros finden Sie unter Beschreibung der Anweisung **DRAW** im *BASIC-Befehlsverzeichnis*.

### Beispiel

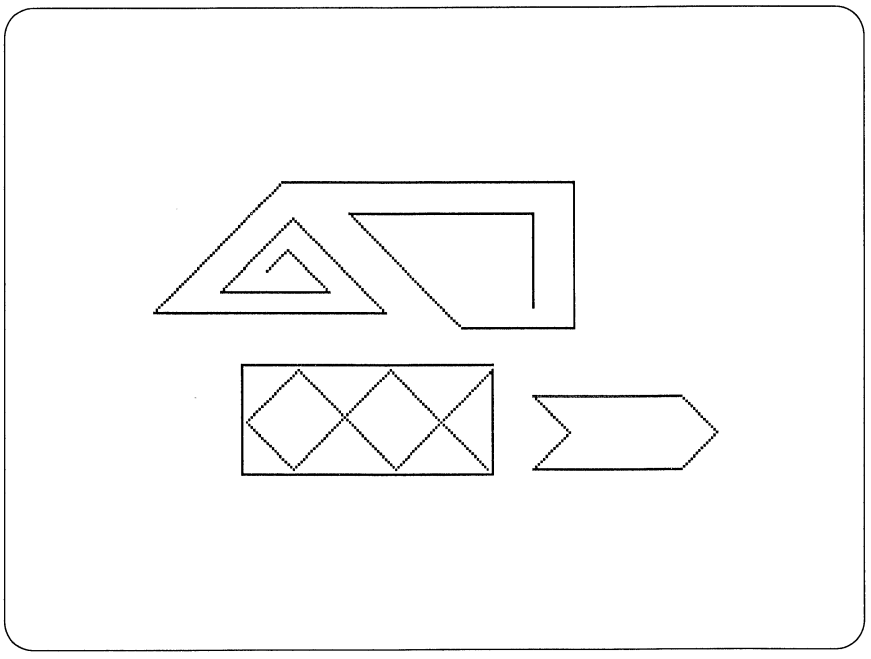
Das folgende Programm gibt eine kurze Einführung in die Bewegungsmakros **U, D, L, R, E, F, G** und **H**; den Makro "zeichne/zeichne nicht", **B**; und den Farbmakro **C**. Dieses Programm zeichnet horizontale, vertikale und diagonale Geraden in unterschiedlichen Farben davon abhängig, welche RICHTUNGSTASTEN auf dem Zehnerblock betätigt werden (↑, →, ↓, BILD ↑, BILD ↓ etc.).

Dieses Programm befindet sich in der **PLOTTER.BAS** benannten Datei auf Ihren QuickBASIC-Originaldisketten.

```
' Werte für Tasten auf dem Zehnerblock und die
' Leertaste:
CONST OBEN = 72, UNTEN = 80, LI = 75, RE = 77
CONST OBENLI = 71, OBENRE = 73, UNTLI = 79, UNTRE = 81
CONST LEERZEICH = " "

' Null$ ist das erste Zeichen des Zweizeichen-Wertes,
' den INKEY$ für Richtungstasten wie OBEN und UNTEN
' zurückgibt:
Null$ = CHR$(0)

' Zeichne$ = "" bedeutet zeichne Geraden;
' Zeichne$ = "B" bedeutet bewege den grafischen
' Cursor, aber zeichne keine Geraden:
Zeichne$ = ""

PRINT "Benutzen Sie die Richtungstasten, um Geraden _
 zu zeichnen."
PRINT "Betätigen Sie die Leertaste, um das Zeichnen _
 von Geraden ein- bzw. auszuschalten."
PRINT "Betätigen Sie <EINGABETASTE>, um zu beginnen."
PRINT "Betätigen Sie E zum Beenden des Programmes."
DO : LOOP WHILE INKEY$ = ""

SCREEN 1

DO
   SELECT CASE TastenWert$
      CASE Null$ + CHR$(OBEN)
         DRAW Zeichne$ + "C1 U2"
      CASE Null$ + CHR$(UNTEN)
         DRAW Zeichne$ + "C1 D2"
      CASE Null$ + CHR$(LI)
         DRAW Zeichne$ + "C2 L2"
      CASE Null$ + CHR$(RE)
         DRAW Zeichne$ + "C2 R2"
      CASE Null$ + CHR$(OBENLI)
         DRAW Zeichne$ + "C3 H2"
      CASE Null$ + CHR$(OBENRE)
         DRAW Zeichne$ + "C3 E2"
      CASE Null$ + CHR$(UNTLI)
         DRAW Zeichne$ + "C3 G2"
      CASE Null$ + CHR$(UNTRE)
         DRAW Zeichne$ + "C3 F2"
```

```
      CASE LEERZEICH
         IF Zeichne$ = "" THEN
            Zeichne$ = "B "
         ELSE
            Zeichne$ = ""
         END IF
      CASE ELSE
         ' Der Benutzer betätigte eine andere Taste
         ' als eine der Richtungstasten, der Leertaste
         ' oder "E", also tue nichts.
   END SELECT

   TastenWert$ = INKEY$
LOOP UNTIL TastenWert$ = "E"
SCREEN 0, 0       ' Stelle den 80-Spalten Textmodus
                  ' wieder her und beende.
WIDTH 80
END
```

### *Ausgabe*



## 5.10 Grundlegende Techniken der Animation

Wenn nur die in den vorherigen Abschnitten behandelten Grafikanweisungen verwendet werden, können Sie einfache Animation mit Objekten auf dem Bildschirm ausführen. Zum Beispiel können Sie zunächst mit **CIRCLE** einen Kreis malen, diesen anschließend mit der Hintergrundfarbe erneut zeichnen, um ihn zu löschen, und zum Schluß den Mittelpunkt des Kreises erneut berechnen, um diesen an einer neuen Stelle zu zeichnen.

Diese Technik reicht für einfache Abbildungen aus, jedoch werden deren Nachteile beim Aufbau komplexerer Bilder offensichtlich. Obwohl die Grafikanweisungen sehr schnell sind, können Sie noch Verzögerungen beobachten. Darüberhinaus ist es mit dieser Methode nicht möglich, den Hintergrund beizubehalten: Wenn Sie das Objekt entfernen, entfernen Sie alles, was sich bereits auf dem Bildschirm befand.

Zwei Anweisungen ermöglichen es Ihnen, schnelle Animation durchzuführen: **GET** und **PUT**. Sie können ein Bild erzeugen, indem Sie Ausgaben von Anweisungen wie **PSET**, **LINE**, **CIRCLE** oder **PAINT** verwenden, dann einen "Schnappschuß" des Bildes mit **GET** machen, wodurch das Bild in den Speicher kopiert wird. Sie können dann mit **PUT** das mit **GET** gespeicherte Bild an beliebiger Stelle auf dem Bildschirm bzw. Darstellungsfeld reproduzieren.

### 5.10.1 Wie Sie Bilder mit GET speichern

Nachdem Sie das Originalbild auf dem Bildschirm erzeugt haben, müssen Sie die *x*- und *y*-Koordinaten eines Rechteckes genügend groß berechnen, um das gesamte Bild aufzunehmen. Dann können Sie **GET** dazu verwenden, das gesamte Rechteck in den Speicher zu kopieren. Die Schreibweise der grafischen Anweisung **GET** ist

**GET** [**STEP**] (*x1*,*y1*) - [**STEP**] (*x2*,*y2*), *Datenfeldname*

wobei (*x1*,*y1*) und (*x2*,*y2*) die übliche Möglichkeit angeben, in BASIC ein Rechteck festzulegen: als Koordinaten seiner oberen linken bzw. unteren rechten Ecke. *Datenfeldname* ist ein beliebiges numerisches Datenfeld. Die in einer **DIM**-Anweisung für *Datenfeldname* angegebene Größe hängt von den folgenden drei Faktoren ab:

1. der Höhe und Breite des Rechtecks, das das Bild einschließt
2. dem Bildschirmmodus, der für grafische Ausgabe gewählt ist
3. dem Typ des Datenfeldes (Ganzzahl, lange Ganzzahl, einfache Genauigkeit oder doppelte Genauigkeit)

***Hinweis*** Obwohl das zum Speichern von Bildern verwendete Datenfeld jeden numerischen Typ haben kann, ist es sehr empfehlenswert, daß Sie nur ganzzahlige Datenfelder verwenden. Alle grafischen Muster, die auf dem Bildschirm möglich sind, können mit Ganzzahlen dargestellt werden. Dies ist jedoch nicht der Fall bei reellen Zahlen einfacher oder doppelter Genauigkeit: Einige grafische Muster sind keine gültigen reellen Zahlen, und deren Speicherung in einem Datenfeld reeller Zahlen könnte zu unvorhergesehenen Ergebnissen führen.

Die Formel zur Berechnung der Größe von *Datenfeldname* in Bytes ist

*Größe in Bytes* = **4** + *Höhe* x *Ebenen* x **INT**((*Breite* x *Bits pro Bildpunkt/Ebenen* + **7**)/**8**)

wobei *Höhe* und *Breite* die Dimensionen in Anzahl an Bildpunkten des zu erhaltenden Rechteckes sind; die *Bits pro Bildpunkt* hängen von der Anzahl der in dem gegebenen Bildschirmmodus verfügbaren Farben ab. Die Regel lautet: mehr Farben bedeutet, zur Definition jedes Bildpunktes werden mehr Bits benötigt. Im Bildschirmmodus 1 definieren zwei Bits einen Bildpunkt, während im Bildschirmmodus 2 ein Bit ausreicht. (Die allgemeine Formel für Bits pro Bildpunkt finden Sie in Abschnitt 5.8.2.1.) Die folgende Liste zeigt die Werte für *Ebenen* in jedem Bildschirmmodus:

| ***Bildschirmmodi*** | ***Anzahl von Bit-Ebenen*** |
|---|---|
| 1, 2, 11 und 13 | 1 |
| 9 (64K Videospeicher) und 10 | 2 |
| 7, 8, 9 (mehr als 64K Videospeicher) und 12 | 4 |

Um die Anzahl von Elementen, die das Datenfeld haben sollte, zu erhalten, dividieren Sie *Größe in Bytes* durch die Anzahl an Bytes für ein Element in dem Datenfeld. Dies ist der Punkt, an dem der Typ des Datenfeldes ins Spiel kommt. Wenn es ein ganzzahliges Datenfeld ist, belegt jedes Element zwei Bytes im Speicher (die Größe einer Ganzzahl), so daß *Größe in Bytes* durch zwei dividiert werden sollte, um die tatsächliche Größe des Datenfeldes zu bekommen. Wenn das Datenfeld den Typ lange Ganzzahl hat, sollte *Größe in Bytes* durch vier dividiert werden (da eine lange Ganzzahl im Speicher vier Bytes beansprucht) usw. Wenn es ein Datenfeld des Typs einfacher Genauigkeit ist, dividieren Sie durch vier; wenn es doppelter Genauigkeit ist, dividieren Sie durch acht.

Die folgenden Schritte zeigen, wie die Größe eines ganzzahligen Datenfeldes berechnet wird, das groß genug ist, ein Rechteck mit den Koordinaten `(10,40)` für die obere linke Ecke und `(90,80)` für die untere rechte Ecke im Bildschirmmodus 1 aufzunehmen:

1. Berechnen Sie die Höhe und Breite des Rechtecks:

```
RechtHoehe  = ABS(y2 - y1) + 1 = 80 - 40 + 1 = 41
RechtBreite = ABS(x2 - x1) + 1 = 90 - 10 + 1 = 81
```

Vergessen Sie nicht, nach der Subtraktion `y1` von `y2` bzw. `x1` von `x2` eine Eins hinzuzuaddieren. Wenn z. B. `x1` gleich 10 und `x2` gleich 20 ist, dann ist die Breite des Rechtecks 20 - 10 + 1 oder 11.

2. Berechnen Sie die Größe des Ganzzahldatenfeldes in Bytes:

```
ByteGroesse = 4 + RechtHoehe *
              INT((RechtBreite * BitsProPixel + 7) / 8)
            = 4 + 41 * INT(( 81 * 2 + 7) / 8)
            = 4 + 41 * INT(169/8)
            = 4 + 41 * 21
            = 865
```

3. Dividieren Sie die Größe in Bytes durch die Bytes pro Element (zwei für Ganzzahlen) und runden Sie das Ergebnis auf die nächste Ganzzahl auf:

```
865/2 = 433
```

Wenn der Name des Datenfeldes `Bild` ist, stellt daher die folgende **DIM**-Anweisung sicher, daß `Bild` genügend groß ist, die Information der Bildpunkte des Rechtecks zu kopieren:

```
DIM Bild (1 TO 433) AS INTEGER
```

***Hinweis*** Obwohl die **GET**-Anweisung nach einer **WINDOW**-Anweisung logische Koordinaten verwendet, müssen Sie zur Berechnung der Größe des in **GET** verwendeten Datenfeldes physikalische Koordinaten benutzen. (Weitere Informationen zu **WINDOW** und zur Umwandlung logischer Koordinaten in physikalische Koordinaten finden Sie in Abschnitt 5.6, "Wie Sie die Koordinaten eines Darstellungsfeldes mit **WINDOW** neu definieren".)

Beachten Sie, daß die oben angegebenen Schritte die für das Datenfeld erforderliche Mindestgröße angeben; jedoch jede Größe darüberhinaus funktioniert ebenfalls. Zum Beispiel arbeitet auch die folgende Anweisung:

```
DIM Bild (1 TO 500) AS INTEGER
```

### Beispiel

Das folgende Programm zeichnet eine Ellipse und malt deren Inneres aus. Eine **GET**-Anweisung kopiert den rechteckigen Bereich, der die Ellipse enthält, in den Speicher. (Abschnitt 5.10.2 zeigt, wie Sie die Anweisung **PUT** verwenden, um die Ellipse an einer anderen Stelle zu reproduzieren.)

```
SCREEN 1
' Dimensioniere ein ganzzahliges Datenfeld, das
' genügend groß ist, das Rechteck aufzunehmen:
DIM Bild (1 TO 433) AS INTEGER
' Zeichne innerhalb des Rechteckes eine Ellipse,
' verwende violett für die Umrandung und male das
' Innere weiß aus:
CIRCLE (50,60),40,2,,,.5
PAINT (50,60), 3, 2
' Speichere das Abbild des Rechtecks in dem Datenfeld:
GET (10,40) - (90,80), Bild
```

## 5.10.2 Wie Sie Bilder mit PUT bewegen

Während es Ihnen die **GET**-Anweisung erlaubt, eine Momentaufnahme eines Bildes zu erstellen, können Sie mit der Anweisung **PUT** das Bild an jedem von Ihnen gewünschten Ort auf dem Bildschirm einfügen. Eine Anweisung der Form

**PUT** (*x*,*y*), *Datenfeldname* [, *Aktionswort*]

kopiert das in *Datenfeldname* gespeicherte rechteckige Bild zurück auf den Bildschirm und plaziert dieses mit der oberen linken Ecke an dem Punkt mit den Koordinaten (*x*,*y*). Beachten Sie, daß in **PUT** nur ein Koordinatenpaar erscheint.

***Hinweis*** Wenn in dem Programm vor **PUT** eine **WINDOW**-Anweisung erscheint, beziehen sich die Koordinaten *x* und *y* auf die untere linke Ecke des Rechtecks.

**WINDOW SCREEN** hat jedoch nicht diese Auswirkung; d. h., nach **WINDOW SCREEN** beziehen sich *x* und *y* nach wie vor auf die obere linke Ecke des Rechtecks.

Zum Beispiel bewirkt das Hinzufügen der folgenden Zeile zu dem vorhergehenden Beispiel, daß eine exakte Kopie der ausgemalten Ellipse auf der rechten Seite des Bildschirms erscheint; viel schneller als wenn dieselbe Abbildung mit **CIRCLE** und **PAINT** neu gezeichnet und neu ausgemalt wird:

```
PUT (200,40), Bild
```

***Wichtig*** Achten Sie darauf, keine Koordinaten anzugeben, die irgendeinen Teil des Abbildes außerhalb des Bildschirms oder aktiven Darstellungsfeldes positionieren, wie in dem folgenden Beispiel:

```
SCREEN 2
.
.
.
'Das Rechteck mißt 141 x 91 Bildpunkte:
GET (10,10) - (150,100), Bild
PUT (510,120), Bild
```

Anders als andere Grafikanweisungen, wie z. B. **LINE** oder **CIRCLE**, schneidet **PUT** außerhalb des Darstellungsfeldes liegende Bilder nicht ab. Stattdessen produziert es die Fehlermeldung `Unzulässiger Funktionsaufruf.`

Einer der weiteren Vorteile der **PUT**-Anweisung ist, daß Sie mit dem Argument *Aktionswort* steuern können, wie sich das gespeicherte Abbild und das, was sich bereits auf dem Bildschirm befindet, gegenseitig beeinflussen. Das Argument *Aktionswort* kann eine der folgenden Optionen sein: **PSET, PRESET, AND, OR** oder **XOR.**

Wenn es Ihnen gleich ist, was mit dem vorhandenen Bildschirmhintergrund geschieht, verwenden Sie die Option **PSET**, da diese eine exakte Kopie des gespeicherten Abbildes auf den Bildschirm überträgt, und alles überschreibt, was bereits dort vorhanden ist.

Die folgende Tabelle zeigt, wie andere Optionen der Anweisung **PUT** festlegen, wie sich Bildpunkte eines gespeicherten Bildes und Bildpunkte auf dem Bildschirm gegenseitig beeinflussen. In dieser Tabelle bedeutet eine 1, ein Bildpunkt ist eingeschaltet, und eine 0 bedeutet, ein Bildpunkt ist ausgeschaltet.

| ***Funktionsoption*** | ***Bildpunkt im gespeicherten Bild*** | ***Bildpunkt auf dem Bildschirm vor der Anweisung PUT*** | ***Bildpunkt auf dem Bildschirm nach der Anweisung PUT*** |
|---|---|---|---|
| **PSET** | 0 | 0 | 0 |
| | 0 | 1 | 0 |
| | 1 | 0 | 1 |
| | 1 | 1 | 1 |
| **PRESET** | 0 | 0 | 1 |
| | 0 | 1 | 1 |
| | 1 | 0 | 0 |
| | 1 | 1 | 0 |

| *Funktionsoption* | *Bildpunkt im gespeicherten Bild* | *Bildpunkt auf dem Bildschirm vor der Anweisung PUT* | *Bildpunkt auf dem Bildschirm nach der Anweisung PUT* |
|---|---|---|---|
| **AND** | 0 | 0 | 0 |
| | 0 | 1 | 0 |
| | 1 | 0 | 0 |
| | 1 | 1 | 1 |
| **OR** | 0 | 0 | 0 |
| | 0 | 1 | 1 |
| | 1 | 0 | 1 |
| | 1 | 1 | 1 |
| **XOR** | 0 | 0 | 0 |
| | 0 | 1 | 1 |
| | 1 | 0 | 1 |
| | 1 | 1 | 0 |

Wie Sie sehen können, veranlassen diese Optionen eine **PUT**-Anweisung, Bildpunkte auf dieselbe Art zu behandeln, wie logische Operatoren Zahlen behandeln. **PRESET** ist darin dem logischen Operator **NOT** gleich, daß es die Bildpunkte des gespeicherten Bildes umkehrt, ungeachtet dessen, was sich auf dem Bildschirm befindet. Die Optionen **AND**, **OR** und **XOR** sind identisch mit den logischen Operatoren gleichen Namens; zum Beispiel hat die logische Operation

```
1 XOR 1
```

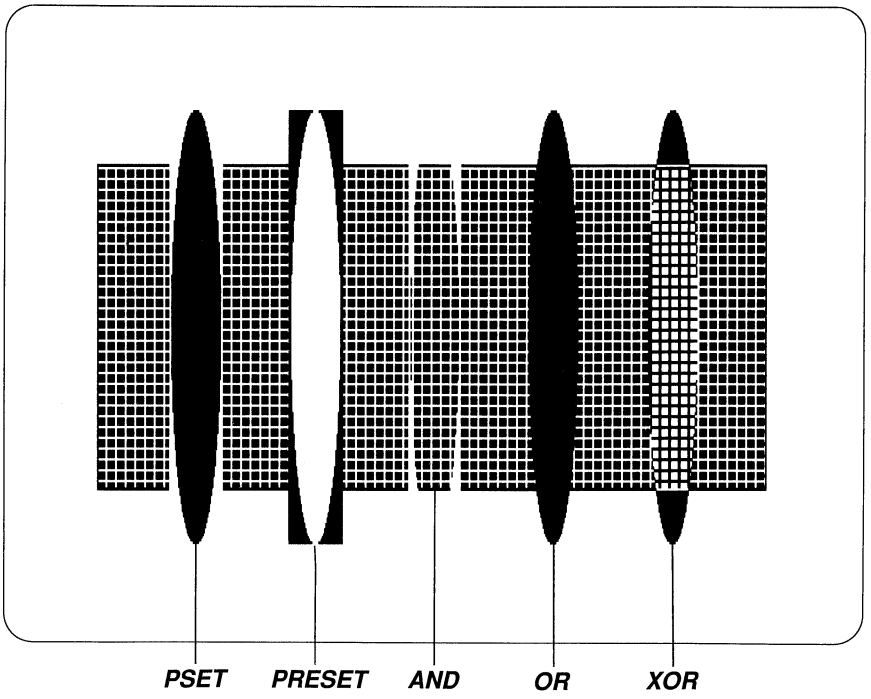
0 als ihr Ergebnis, genauso wie die Verwendung der Option **XOR** einen Bildpunkt ausschaltet, wenn sowohl der Bildpunkt des Abbildes als auch der Bildpunkt des Hintergrundes eingeschaltet sind. (Weitere Informationen zu logischen Operatoren finden Sie in Kapitel 3, "Ausdrücke und Operatoren", im *BASIC-Befehlsverzeichnis*.)

Die Ausgabe des folgenden Programmes, die dasselbe Bild zeigt, das unter Verwendung jeder der fünf oben erläuterten Optionen einem ausgefüllten Rechteck überlagert wird, soll helfen, diese Auswirkungen zu verdeutlichen:

```
SCREEN 2
DIM EllipsBild (1 TO 485) AS INTEGER
' Zeichne und male eine Ellipse aus, speichere dann
' ihr Abbild mit GET:
CIRCLE (22,100),80,,,,4
PAINT (22,100)
```

```
GET (0,20) - (44,180), EllipsBild
CLS
' Zeichne ein Rechteck und fülle es mit einem Muster:
LINE (40,40) - (600,160),,B
Muster$ = CHR$(126) + CHR$(0) + CHR$(126) + CHR$(126)
PAINT (50,50),Muster$
' Lege die Abbilder der Ellipse über das Rechteck
' unter Verwendung unterschiedlicher Funktionsoptionen:
PUT (100,20), EllipsBild, PSET
PUT (200,20), EllipsBild, PRESET
PUT (300,20), EllipsBild, AND
PUT (400,20), EllipsBild, OR
PUT (500,20), EllipsBild, XOR
```

***Ausgabe***



Mit Bildschirmmodi, die Farbe unterstützen, bewirken die Optionen **PRESET**, **AND**, **OR** und **XOR** kompliziertere Wechselwirkungen, da Farbe mehr beinhaltet als einfaches Ein- oder Ausschalten eines Bildpunktes. Die oben hergestellte Analogie zwischen diesen Optionen und logischen Operatoren gilt jedoch auch in diesen Modi. Wenn zum Beispiel der aktuelle Bildpunkt auf dem Bildschirm die Farbe 1 (kobaltblau in Palette 1) hat, und der Bildpunkt des überlagerten Bildes die Farbe 2 (violett in Palette 1) hat, dann hängt die Farbe des nach einer **PUT**-Anweisung sich ergebenden Bildpunktes von der jeweiligen Option ab, wie in der folgenden Tabelle für nur 2 der 16 unterschiedlichen Kombinationen von Bildfarbe und Hintergrundfarbe gezeigt:

| *Funktionsoption* | *Bildpunktfarbe im gespeicherten Bild* | *Bildpunktfarbe auf dem Bildschirm vor der Anweisung PUT* | *Bildschirmfarbe auf dem Bildschirm nach der Anweisung PUT* |
|---|---|---|---|
| **PSET** | 10 (violett) | 01 (kobaltblau) | 10 (violett) |
| **PRESET** | 10 (violett) | 01 (kobaltblau) | 01 (kobaltblau) |
| **AND** | 10 (violett) | 01 (kobaltblau) | 00 (schwarz) |
| **OR** | 10 (violett) | 01 (kobaltblau) | 11 (weiß) |
| **XOR** | 10 (violett) | 01 (kobaltblau) | 11 (weiß) |

In Palette 1 ist kobaltblau dem Attribut 1 (01 binär) zugewiesen, violett ist dem Attribut 2 (10 binär) zugewiesen, und weiß ist dem Attribut 3 (11 binär) zugewiesen. Wenn Sie eine EGA besitzen, können Sie die Anweisung **PALETTE** verwenden, um den Attributen 1, 2 und 3 unterschiedliche Farben zuzuweisen.

## 5.10.3 Animation mit GET und PUT

Besonders nützlich sind die Anweisungen **GET** und **PUT** für Animation. Die beiden am besten geeigneten Optionen für Animation sind **XOR** und **PSET**. Mit **PSET** ausgeführte Animation ist zwar schneller, löscht aber, wie mit der Ausgabe des letzten Programmes gezeigt, den Bildschirmhintergrund. **XOR** dagegen ist zwar langsamer, stellt den Bildschirmhintergrund aber nach der Bewegung des Bildes wieder her. Animation mit **XOR** geschieht mit den folgenden vier Schritten:

1. Bringen Sie das Objekt mit **XOR** auf den Bildschirm.
2. Berechnen Sie die neue Position des Objektes.
3. Bringen Sie das Objekt ein zweitesmal an der alten Stelle auf den Bildschirm, verwenden Sie wiederum **XOR**, diesmal zum Entfernen des alten Bildes.
4. Gehen Sie zurück zu Schritt 1, bringen Sie aber diesmal das Objekt an die neue Position.

Die mit diesen vier Schritten durchgeführte Bewegung läßt den Hintergrund nach Schritt 3 unverändert. Bildflimmern kann verringert werden, wenn die Zeit zwischen den Schritten 4 und 1 minimiert und sichergestellt wird, daß genügend Verzögerungszeit zwischen den Schritten 1 und 3 vorhanden ist. Wenn mehr als 1 Objekt aufgebaut werden soll, sollte jedes Objekt auf einmal verarbeitet werden, Schritt für Schritt.

Wenn es nicht notwendig ist, den Hintergrund beizubehalten, verwenden Sie für Animation die Option **PSET**. Der Grundgedanke ist, einen Rahmen um das Bild herum zu belassen, wenn Sie es mit der Anweisung **GET** kopieren. Wenn dieser Rahmen so groß oder größer ist als der maximale Abstand, um den das Objekt sich bewegt, dann entfernt der Rahmen jedesmal, wenn das Bild an eine neue Stelle gesetzt wird, alle Spuren des Bildes an der alten Position. Diese Methode kann um einiges schneller sein als die oben beschriebene Methode, die **XOR** verwendet, da für die Bewegung eines Objektes nur eine **PUT**-Anweisung erforderlich ist (obwohl Sie ein größeres Bild bewegen müssen).

### Beispiel

Das folgende Beispiel zeigt, wie **PUT** mit der Option **PSET** verwendet wird, um den Effekt eines von dem unteren Rand und den seitlichen Rändern eines Rechtecks zurückprallenden Balles zu erzeugen. Beachten Sie in der folgenden Ausgabe, wie das den Ball enthaltene Rechteck, das ausgefüllte Rechteck und die geschriebene Meldung löscht.

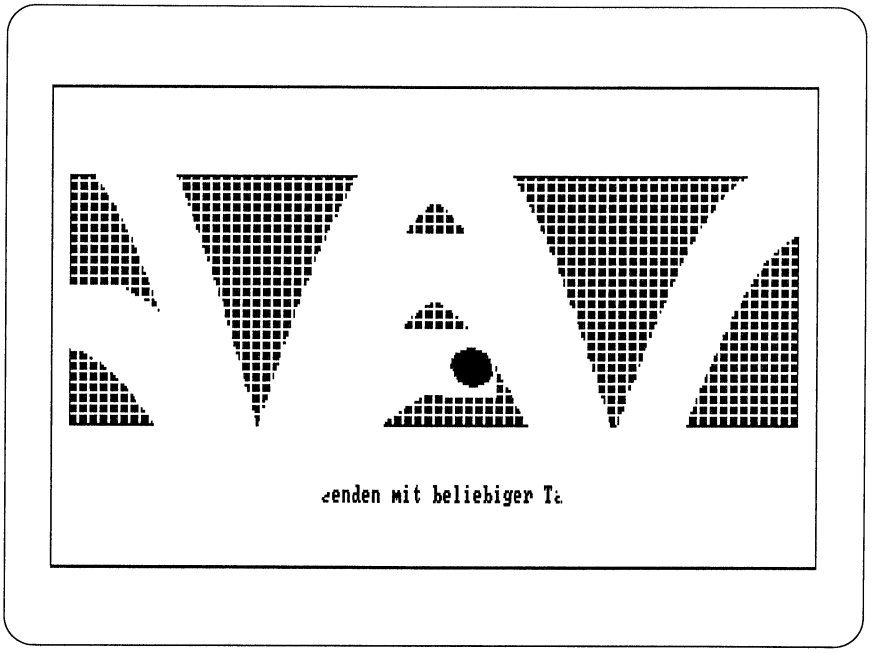
Dieses Programm befindet sich in der **BALLPSET.BAS** benannten Datei auf Ihren QuickBASIC-Originaldisketten.

```
DECLARE FUNCTION FeldGroesse(WLinks, WRechts, WOben, WUnten)
SCREEN 2
' Definiere ein Darstellungsfeld und zeichne hierum
' einen Rahmen:
VIEW (20,10) - (620,190),,1
CONST PI = 3.141592653589#
' Definiere die Koordinaten des Darstellungsfeldes neu
' mit logischen Koordinaten:
WINDOW (-3.15,-.14) - (3.56,1.01)
' Die Datenfelder im Programm sind nun dynamisch:
' $DYNAMIC
' Berechne die logischen Koordinaten für den oberen
' und unteren Rand eines Rechtecks genügend groß, um das
' Bild aufzunehmen, das mit CIRCLE und PAINT gezeichnet
' wird.
```

```
WLinks   = -.21
WRechts  =  .21
WOben    =  .07
WUnten   = -.07

' Rufe die Funktion FeldGroesse auf und übergib ihr
' die logischen Koordinaten des Rechtecks:
FeldGross% = FeldGroesse(WLinks, WRechts, WOben, WUnten)

DIM Feld (1 TO FeldGross%) AS INTEGER

' Zeichne und male den Kreis aus:
CIRCLE (0,0),.18
PAINT (0,0)

' Speichere das Rechteck in Feld:
GET (WLinks,WOben) - (WRechts,WUnten), Feld
CLS

' Zeichne ein Rechteck und fülle es mit einem Muster:
LINE (-3,.8) - (3.4,.2),,B
Muster$ = CHR$(126) + CHR$(0) + CHR$(126) + CHR$(126)
PAINT (0,.5),Muster$

LOCATE 21, 29
PRINT "Beenden mit beliebiger Taste"

' Initialisiere die Schleifenvariablen:
SchrWeit      = .02
StartSchleife = -PI
NimmAb        =  1

DO
   EndSchleife = -StartSchleife
   FOR X = StartSchleife TO EndSchleife STEP SchrWeit

      ' Jedesmal wenn der Ball "aufspringt" (auf dem
      ' unteren Rand des Darstellungsfeldes
      ' auftrifft), wird die Variable NimmAb kleiner,
      ' wodurch der nächste Aufstieg kleiner wird:
      Y = ABS(COS(X)) * NimmAb - .14
      IF Y < -.13 THEN NimmAb = NimmAb * .9

      ' Beende nach Tastenbetätigung oder wenn NimmAb
      ' kleiner als .01 ist:
      Esc$ = INKEY$
      IF Esc$ <> "" OR NimmAb < .01 THEN EXIT FOR
```

```
         ' Bringe das Bild auf den Bildschirm. Der Offset
         ' SchrWeit ist kleiner als der Rahmen um den
         ' Kreis herum. Daher löscht das Bild jedesmal,
         ' wenn es sich bewegt, alle vom vorhergehenden
         ' PUT zurückgelassenen Spuren (und löscht ebenso
         ' alles andere auf dem Bildschirm):
         PUT (X,Y), Feld, PSET
      NEXT X

      ' Kehre Richtung um:
      SchrWeit  = -SchrWeit
      StartSchleife = -StartSchleife
LOOP UNTIL Esc$ <> "" OR NimmAb < .01
END


FUNCTION FeldGroesse(WLinks, WRechts, WOben, WUnten) STATIC
      ' Bilde die an diese Funktion übergebenen logischen
      ' Koordinaten auf deren entsprechende physikalische
      ' Koordinaten ab:
      VLinks   = PMAP(WLinks,  0)
      VRechts  = PMAP(WRechts, 0)
      VOben    = PMAP(WOben,   1)
      VUnten   = PMAP(WUnten,  1)

      ' Berechne die Höhe und Breite des umschließenden
      ' Rechtecks in Bildpunkten:
      RechtHoehe = ABS(VUnten - VOben) + 1
      RechtBreit = ABS(VRechts - VLinks) + 1

      ' Berechne Größe des Datenfeldes in Bytes:
      ByteGross = 4 + RechtHoehe * INT((RechtBreit + 7) / 8)

      ' Datenfeld ist Ganzzahl, also dividiere Bytes
      ' durch 2:
      FeldGroesse = ByteGross \ 2 + 1
END FUNCTION
```

### Ausgabe

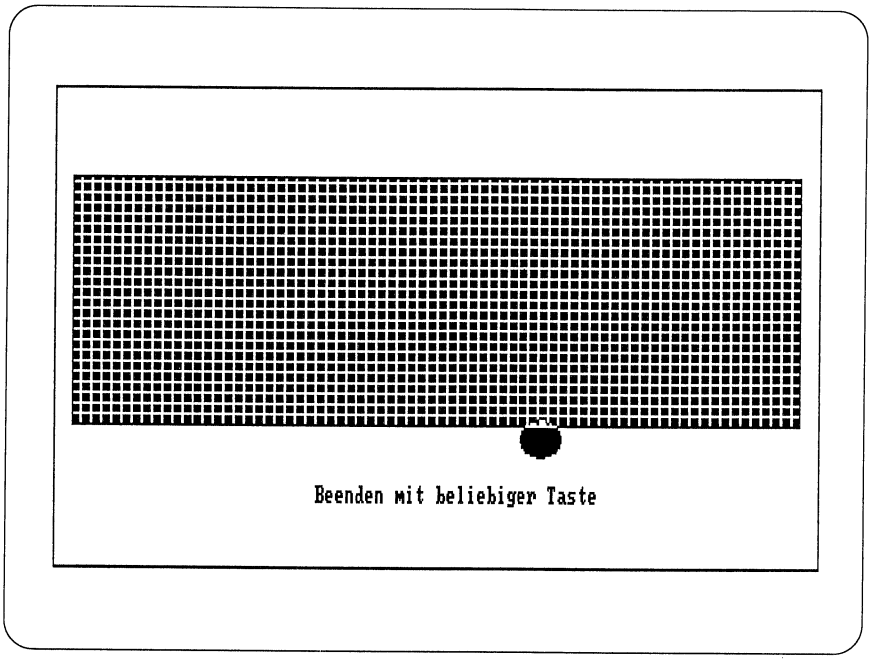



Vergleichen Sie das vorhergehende Programm mit dem nächsten Programm, das **PUT** mit **XOR** verwendet, um den Hintergrund des Bildschirms beizubehalten, wie in den vier oben aufgeführten Schritten beschrieben wurde. Beachten Sie, daß das Rechteck, das den Ball enthält, kleiner als im vorhergehenden Programm ist, da es nicht notwendig ist, einen Rahmen zu lassen. Beachten Sie auch, daß zwei **PUT**-Anweisungen erforderlich sind, eine, um das Bild sichtbar zu machen, und eine weitere, um es verschwinden zu lassen. Beachten Sie außerdem die leere **FOR...NEXT**-Schleife zwischen den **PUT**-Anweisungen; diese Schleife verringert das Flimmern, das seine Ursache im zu schnellen Erscheinen und Verschwinden des Bildes hat.

Dieses Programm befindet sich in der **BALLXOR.BAS** benannten Datei auf Ihren QuickBASIC-Originaldisketten.

```
.
.
.
' Das Rechteck ist kleiner als das im vorhergehenden
' Programm, daher ist auch Feld kleiner:
WLinks   = -.18
WRechts  =  .18
WOben    =  .05
WUnten   = -.05
.
.
.
DO
   EndSchleife = -StartSchleife
   FOR X = StartSchleife TO EndSchleife STEP SchrWeit
      Y = ABS(COS(X)) * NimmAb - .14
      ' Die erste PUT-Anweisung plaziert das Bild auf
      ' dem Bildschirm:
      PUT (X,Y), Feld, XOR
      ' Verwende eine leere FOR...NEXT-Schleife, um
      ' das Programm zu verzögern und das Bildflimmern
      ' zu verringern:
      FOR I = 1 TO 5 : NEXT I
      IF Y < -.13 THEN NimmAb = NimmAb * .9
      Esc$ = INKEY$
      IF Esc$ <> "" OR NimmAb < .01 THEN EXIT FOR
      ' Die zweite PUT-Anweisung löscht das Bild und
      ' stellt den Hintergrund wieder her:
      PUT (X,Y), Feld, XOR
   NEXT X
   SchrWeit  = -SchrWeit
   StartSchleife = -StartSchleife
LOOP UNTIL Esc$ <> "" OR NimmAb < .01
END
.
.
.
```

***Ausgabe***




## 5.10.4 Animation mit Bildschirmseiten

Dieser Abschnitt beschreibt eine Technik der Animation, die mehrere Seiten des Videospeichers Ihres Computers ausnutzt.

Seiten im Videospeicher sind vergleichbar mit Seiten in einem Buch. Abhängig von den grafischen Fähigkeiten Ihres Computers kann das, was Sie auf dem Bildschirm sehen, nur ein Teil des verfügbaren Videospeichers sein - genauso wie das, was Sie nach Aufschlagen eines Buches sehen, nur ein Teil des Buches ist. Anders als bei einem Buch können die unsichtbaren Seiten des Videospeichers Ihres Computers jedoch aktiv sein; das heißt, während Sie sich eine Seite auf dem Bildschirm ansehen, kann grafische Ausgabe auf den anderen stattfinden. Das ist so, als wenn der Autor eines Buches noch neue Seiten schreibt, während Sie das Buch bereits lesen.

Der auf dem Bildschirm sichtbare Bereich des Videospeichers wird als "sichtbare Seite" bezeichnet, während der Bereich des Videospeichers, in den grafische Anweisungen ihre Ausgabe schreiben, als "aktive Seite" bezeichnet wird. Die **SCREEN**-Anweisung erlaubt es Ihnen, mit der folgenden Schreibweise die sichtbaren bzw. aktiven Bildschirmseiten auszuwählen:

**SCREEN** [*Modus*],[,[*A-Seite*][,*S-Seite*]]

In dieser Schreibweise ist *A-Seite* die Nummer der aktiven Seite und *S-Seite* die Nummer der sichtbaren Seite. (Umfassende Informationen zu den Bildschirmmodi, die mehrere Seiten unterstützen, und den zulässigen Bereichen für Seitenargumente finden Sie in der Beschreibung der **SCREEN**-Anweisung im *BASIC-Befehlsverzeichnis*.)

Die aktive Seite und die sichtbare Seite können ein und dasselbe sein (und sind es standardmäßig, wenn die Argumente *A-Seite* und *S-Seite* nicht mit **SCREEN** verwendet werden; in dem Fall sind die Werte beider Argumente 0).

Sie können Objekte auf dem Bildschirm bewegen, indem Sie einen Bildschirmmodus mit mehr als einer Seite Videospeicher auswählen, dann die Seiten wechseln und Ausgabe auf eine oder mehrere aktive Seiten senden, während eine abgeschlossene Ausgabe auf dem Bildschirm angezeigt wird. Diese Technik läßt eine aktive Seite nur sichtbar werden, wenn die Ausgabe auf dieser Seite komplett ist. Da der Betrachter nur ein abgeschlossenes Bild sieht, erscheint die Anzeige augenblicklich.

### Beispiel

Das folgende Programm demonstriert die oben erläuterte Technik. Es wählt den Bildschirmmodus 7 aus, der zwei Seiten hat, und zeichnet dann mit der Anweisung **DRAW** einen Würfel. Dieser Würfel wird dann nach und nach mit 15° Winkeln gedreht, indem der Wert des **TA**-Makros in der von **DRAW** verwendeten Zeichenkette verändert wird. Durch Hin- und Herschalten der aktiven und sichtbaren Seiten zeigt dieses Programm immer einen vollständigen Würfel, während ein neuer gezeichnet wird.

Dieses Programm befindet sich in der **KUBUS.BAS** benannten Datei auf den QuickBASIC-Originaldisketten.

```
' Definiere die Makro-Zeichenkette, die zum Zeichnen
' des Würfels sowie Ausmalen seiner Seiten verwendet
' wird:
Plot$ = "BR30 BU25 C1 R54 U45 L54 D45 BE20 P1,1 G20 C2 " +_
  "G20 R54 E20 L54 BD5 P2,2 U5 C4 G20 U45 E20 D45 BL5 P4,4"
AktivSeite% = 1    ' Initialisiere Werte für die
SichtSeite% = 0    ' aktiven und sichtbaren Seiten
Winkel%     = 0    ' sowie für den Rotationswinkel.

DO
   ' Zeichne auf der aktiven Seite, während die
   ' sichtbare Seite gezeigt wird.
   SCREEN 7, , AktivSeite%, SichtSeite%
   CLS 1       ' Lösche die aktive Seite.
   ' Drehe den Würfel "Winkel%" Grad:
   DRAW "TA" + STR$(Winkel%) + Plot$
```

```
    ' Winkel% ist ein Vielfaches von 15 Grad:
    Winkel% = (Winkel% + 15) MOD 360
    ' Das Zeichnen ist abgeschlossen, also lasse den
    ' Würfel in seiner neuen Position sichtbar werden,
    ' durch Vertauschung der aktiven mit der sichtbaren
    ' Seite:
    SWAP AktivSeite%, SichtSeite%
LOOP WHILE INKEY$ = ""          ' Eine Tastenbetätigung
                                ' beendet das Programm.
END
```

## 5.11 Anwendungsbeispiel

Die Beispielanwendungen in diesem Kapitel sind ein Balkendiagramm-Generator, ein Programm, das mit unterschiedlichen Farben Punkte in die Mandelbrot-Menge einzeichnet, und ein Editor für Muster.

### 5.11.1 Balkendiagramm-Generator (BALKEN.BAS)

Dieses Programm verwendet zum Zeichnen eines gefüllten Balkendiagramms alle Formen der **LINE**-Anweisung, die in den Abschnitten 5.3.2.1. bis 5.3.2.3 vorgestellt werden. Jeder Balken wird mit einem Muster ausgefüllt, das in einer **PAINT**-Anweisung festgelegt wird.

Die Eingabe des Programmes umfaßt Überschriften für das Diagramm, Bezeichnungen für die *x*- und *y*-Achsen sowie einen Satz von bis zu 5 Bezeichnungen (mit den zugeordneten Werten) für die Balken.

#### Verwendete Anweisungen und Funktionen

Dieses Programm demonstriert die Verwendung der folgenden Grafikanweisungen:

- **LINE**
- **PAINT** (mit einem Muster)
- **SCREEN**

### Programm-Listing

```
' Definiere den Typ für die Titel:
TYPE TitelTyp
   HauptTitel AS STRING * 40
   XTitel AS STRING * 40
   YTitel AS STRING * 18
END TYPE
DECLARE SUB EingabTitel (T AS TitelTyp)
DECLARE FUNCTION Diagramm$ (T AS TitelTyp, Bezeich$(),
Wert!(), N%)
DECLARE FUNCTION EingabDaten% (Bezeich$(), Wert!())
' Variablendeklarationen für Titel und Balkendaten:
DIM Titel AS TitelTyp, Bezeich$(1 TO 5), Wert(1 TO 5)
CONST FALSCH = 0, WAHR = NOT FALSCH
DO
   EingabTitel Titel
   N% = EingabDaten%(Bezeich$(), Wert())
   IF N% <> FALSCH THEN
      NeuDiagr$ = Diagramm$(Titel, Bezeich$(), Wert(), N%)
   END IF
LOOP WHILE NeuDiagr$ = "J"
END
'
' ==================== DIAGRAMM ======================
'   Zeichnet ein Balkendiagramm für die in den
'   Prozeduren EINGABTITEL und EINGABDATEN
'   eingegebenen Daten.
' ====================================================
'
FUNCTION Diagramm$ (T AS TitelTyp, Bezeich$(), Wert(), N%)_
STATIC
   ' Setze die Größe des Diagramms:
   CONST GRAPHOBEN = 24, GRAPHUNTEN = 171
   CONST GRAPHLINKS = 48, GRAPHRECHTS = 624
   CONST YLANG = GRAPHUNTEN - GRAPHOBEN
   ' Berechne maximalen und minimalen Wert:
   YMax = 0
   YMin = 0
   FOR I% = 1 TO N%
      IF Wert(I%) < YMin THEN YMin = Wert(I%)
      IF Wert(I%) > YMax THEN YMax = Wert(I%)
   NEXT I%
```

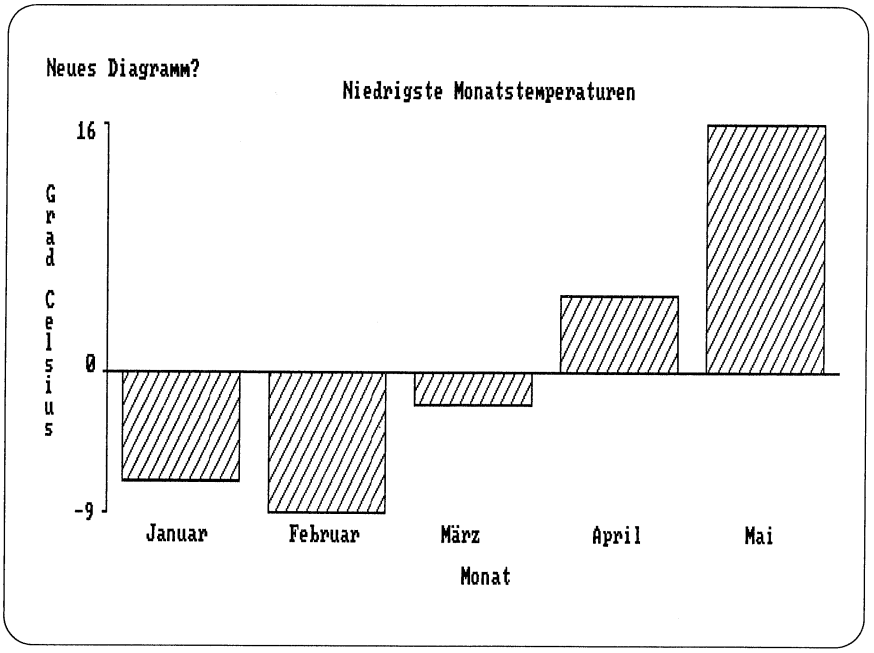
```
' Berechne Breite der Balken und Platz zwischen
' diesen:
BalkBreit = (GRAPHRECHTS - GRAPHLINKS) / N%
BalkZwisch = .2 * BalkBreit
BalkBreit = BalkBreit - BalkZwisch

SCREEN 2
CLS

' Zeichne die y-Achse:
LINE (GRAPHLINKS, GRAPHOBEN)-(GRAPHLINKS, GRAPHUNTEN), 1

' Zeichne Hauptüberschrift des Diagramms:
Start% = 44 - (LEN(RTRIM$(T.HauptTitel)) / 2)
LOCATE 2, Start%
PRINT RTRIM$(T.HauptTitel);

' Bezeichne y-Achse:
Start% = CINT(13 - LEN(RTRIM$(T.YTitel)) / 2)
FOR I% = 1 TO LEN(RTRIM$(T.YTitel))
   LOCATE Start% + I% - 1, 1
   PRINT MID$(T.YTitel, I%, 1);
NEXT I%

' Berechne den Skalierungsfaktor, so daß
' Bezeichnungen nicht größer als 4 Ziffern sind:
IF ABS(YMax) > ABS(YMin) THEN
   Potenz = YMax
ELSE
   Potenz = YMin
END IF
Potenz = CINT(LOG(ABS(Potenz) / 100) / LOG(10))
IF Potenz < 0 THEN Potenz = 0

' Verkleinere minimalen und maximalen Wert:
SkalFaktor = 10 ^ Potenz
YMax = CINT(YMax / SkalFaktor)
YMin = CINT(YMin / SkalFaktor)

' Wenn Potenz nicht Null ist, dann schreibe den
' Skalierungsfaktor in das Diagramm:
IF Potenz <> 0 THEN
   LOCATE 3, 2
   PRINT "x 10^"; LTRIM$(STR$(Potenz))
END IF

' Setze Markierungsstrich und Zahl für maximalen
' Punkt an die y-Achse:
LINE (GRAPHLINKS - 3, GRAPHOBEN)-STEP(3, 0)
LOCATE 4, 2
PRINT USING "####"; YMax
```

```
' Setze Markierungsstrich und Zahl für minimalen
' Punkt an die y-Achse:
LINE (GRAPHLINKS - 3, GRAPHUNTEN) - STEP(3, 0)
LOCATE 22, 2
PRINT USING "####"; YMin

YMax = YMax * SkalFaktor ' Vergrößere Minimum und
                         ' Maximum wieder
YMin = YMin * SkalFaktor ' zur grafischen
                         ' Darstellung der
                         ' Berechnungen.

' Bezeichne x-Achse:
Start% = 44 - (LEN(RTRIM$(T.XTitel)) / 2)
LOCATE 25, Start%
PRINT RTRIM$(T.XTitel);

' Berechne den Bildpunktebereich für die y-Achse:
YBereich = YMax - YMin

' Definiere ein diagonal gestreiftes Muster:
Kachel$ = CHR$(1) + CHR$(2) + CHR$(4) + CHR$(8) + _
          CHR$(16) + CHR$(32) + CHR$(64) + CHR$(128)

' Zeichne eine Null-Linie, falls passend:
IF YMin < 0 THEN
   Unten = GRAPHUNTEN - ((-YMin) / YBereich * YLANG)
   LOCATE INT((Unten - 1) / 8) + 1, 5
   PRINT "0";
ELSE
   Unten = GRAPHUNTEN
END IF

' Zeichne die x-Achse:
LINE (GRAPHLINKS - 3, Unten)-(GRAPHRECHTS, Unten)

' Zeichne die Balken und Bezeichnungen:
Start% = GRAPHLINKS + (BalkZwisch / 2)
FOR I% = 1 TO N%

   ' Zeichne eine Balkenbezeichnung:
   BalkMitt = Start% + (BalkBreit / 2)
   DiagrMitt = INT((BalkMitt - 1) / 8) + 1
   LOCATE 23,DiagrMitt - INT(LEN(RTRIM$(Bezeich$(I%)))/2)
   PRINT Bezeich$(I%);

   ' Zeichne den Balken und fülle ihn mit dem
   ' gestreiften Muster aus:
   BalkHoehe = (Wert(I%) / YBereich) * YLANG
   LINE (Start%, Unten)-STEP(BalkBreit, - BalkHoehe),,B
   PAINT (BalkMitt, Unten - (BalkHoehe / 2)),Kachel$,1
   Start% = Start% + BalkBreit + BalkZwisch
```

```
   NEXT I%
   LOCATE 1, 1
   PRINT "Neues Diagramm? ";
   Diagramm$ = UCASE$(INPUT$(1))
END FUNCTION
'
' ==================== EINGABDATEN ==================
'       Liest Eingabe für die Balkenbezeichnungen
'       und deren Werte
' ===================================================
'
FUNCTION EingabDaten% (Bezeich$(), Wert()) STATIC
   ' Initialisiere die Anzahl der Datenwerte:
   AnzDaten% = 0
   ' Gib Anweisungen für Dateneingabe aus:
   CLS
   PRINT "Geben Sie Daten für bis zu 5 Balken ein:"
   PRINT "   * Geben Sie die Bezeichnung und den Wert";
   PRINT " für jeden Balken ein."
   PRINT "   * Werte können negativ sein."
   PRINT "   * Geben Sie eine leere Bezeichnung zum";
   PRINT " Beenden ein."
   PRINT
   PRINT "Nachdem Sie das Diagramm gesehen haben, ";
   PRINT "betätigen Sie eine Taste"
   PRINT "zum Beenden des Programmes.";
   ' Nimm Daten bis zu einer leeren Bezeichnung
   ' oder 5 Einträgen an:
   Fertig% = FALSCH
   DO
      AnzDaten% = AnzDaten% + 1
      PRINT
      PRINT "Balken("; LTRIM$(STR$(AnzDaten%)); "):"
      INPUT ; "Bezeichnung? ", Bezeich$(AnzDaten%)
      ' Eingabe eines Wertes nur, wenn Bezeichnung
      ' nicht leer ist:
      IF Bezeich$(AnzDaten%) <> "" THEN
         entf = LEN(Bezeich$(AnzDaten%)) + 30
         LOCATE , entf
         INPUT "Wert? ", Wert(AnzDaten%)
```

```
      ' Wenn Bezeichnung leer ist, vermindere den
      ' Datenzähler und setze Flagge für Fertig auf
      ' WAHR:
      ELSE
         AnzDaten% = AnzDaten% - 1
         Fertig% = WAHR
      END IF
   LOOP UNTIL (AnzDaten% = 5) OR Fertig%
   ' Gib die Anzahl der eingegebenen Datenwerte
   ' zurück:
   EingabDaten% = AnzDaten%
END FUNCTION
'
' ================== EINGABTITEL ====================
'       Akzeptiert Eingabe für die drei
'       unterschiedlichen Diagrammtitel
' ===================================================
'
SUB EingabTitel (T AS TitelTyp) STATIC
   SCREEN 0, 0     ' Setze Textbildschirm.
   DO              ' Eingabe der Titel.
      CLS
      INPUT "Geben Sie die Überschrift des Diagramms ein :
", T.HauptTitel
      INPUT "Geben Sie die Bezeichnung der x-Achse ein    :
", T.XTitel
      INPUT "Geben Sie die Bezeichnung der y-Achse ein    :
", T.YTitel
      ' Überprüfe, ob Titel in Ordnung sind:
      LOCATE 7, 1
      PRINT "Eingaben korrekt? J/N ";
      LOCATE , , 1
      OK$ = UCASE$(INPUT$(1))
   LOOP UNTIL OK$ = "J"
END SUB
```

***Ausgabe***




## 5.11.2 Mathematisch erzeugte Farbe in einer Figur (MANDEL.BAS)

Dieses Programm verwendet BASIC-Grafikanweisungen, um eine als "Fractal" bezeichnete Figur zu erzeugen. Ein Fractal ist eine grafische Darstellung dessen, was mit Zahlen geschieht, wenn diese einer wiederholten Folge mathematischer Operationen unterworfen sind. Der von diesem Programm erzeugte Fractal zeigt eine Teilmenge der Menge der sogenannten komplexen Zahlen; diese Teilmenge wird als "Mandelbrot-Menge" bezeichnet, benannt nach Benoit B. Mandelbrot vom IBM Thomas J. Watson Research Center.

Kurz gesagt, bestehen komplexe Zahlen aus zwei Teilen, einem reellen Teil und einem sogenannten imaginären Teil, der ein vielfaches von $\sqrt{-1}$ ist. Quadrieren einer komplexen Zahl, anschließendes Zurückgeben der imaginären und reellen Teile in eine zweite komplexe Zahl und Wiederholung dieses Vorganges veranlaßt einige komplexe Zahlen, sehr schnell ziemlich groß zu werden. Andere Zahlen jedoch läßt dieses Verfahren um einen stabilen Wert schwanken. Die stabilen Werte befinden sich in der Mandelbrot-Menge und werden in diesem Programm durch die Farbe schwarz dargestellt. Die instabilen Werte, also diejenigen, die sich von der Mandelbrot-Menge entfernen, werden durch die anderen Farben aus der Palette dargestellt. Je kleiner das Farbattribut, desto instabiler der Punkt.

Weitere Hintergrundinformation zu der Mandelbrot-Menge finden Sie in dem Aufsatz von A.K. Dewdney' "Computer Recreations", in *Scientific American*, August 1985.

Dieses Programm überprüft ebenfalls das Vorhandensein einer EGA-Karte und zeichnet, wenn diese vorhanden ist, die Mandelbrot-Menge im Bildschirmmodus 8. Nach dem Zeichnen jeder Zeile wechselt das Programm die 16 Farben in der Palette mit einer **PALETTE USING**-Anweisung. Wenn keine EGA-Karte vorhanden ist, zeichnet das Programm eine 4-farbige (weiß, violett, kobaltblau und schwarz) Mandelbrot-Menge im Bildschirmmodus 1.

### Verwendete Anweisungen und Funktionen

Dieses Programm demonstriert die Verwendung der folgenden Grafikanweisungen:

- **LINE**
- **PALETTE USING**
- **PMAP**
- **PSET**
- **SCREEN**
- **VIEW**
- **WINDOW**

### Programm-Listing

```
DEFINT A-Z      ' Standardvariablentyp ist Ganzzahl.
DECLARE SUB AenderPalette ()
DECLARE SUB FenstWerte (FL%, FR%, FO%, FU%)
DECLARE SUB MonitorTest (EM%, FB%, DL%, DR%, DOB%, DU%)
CONST FALSCH = 0, WAHR = NOT FALSCH  ' Boolesche
                                     ' Konstanten
' Lege maximale Anzahl an Iterationen pro Punkt fest:
CONST MAXSCHLEIFE = 30, MAXGROSS = 1000000
DIM PaletteFeld(15)
FOR I = 0 TO 15 : PaletteFeld(I) = I : NEXT I
' Rufe FenstWerte auf, um die Koordinaten der Ecken des
' Fensters zu bekommen:
FenstWerte WLinks, WRechts, WOben, WUnten
' Rufe MonitorTest auf um herauszufinden, ob es sich
' um einen Rechner mit EGA handelt, und lies
' Koordinaten der Ecken des Darstellungsfeldes:
MonitorTest EgaModus, FarbeBereich, VLinks, VRechts, _
            VOben, VUnten
```

```
' Definiere Darstellungsfeld und entsprechendes
' Fenster:
VIEW (VLinks, VOben)-(VRechts, VUnten), 0, FarbeBereich
WINDOW (WLinks, WOben)-(WRechts, WUnten)

LOCATE 24, 10 : PRINT "Beliebige Taste zum Beenden.";

XLaenge = VRechts - VLinks
YLaenge = VUnten - VOben
FarbeBreit = MAXSCHLEIFE \ FarbeBereich

' Bearbeite jeden Bildpunkt im Darstellungsfeld und
' berechne, ob er sich in der Mandelbrot-Menge
' befindet oder nicht:
FOR Y = 0 TO YLaenge    ' Durchlaufe jede Zeile des
                        ' Darstellungsfeldes.
   LogikY = PMAP(Y, 3)  ' Lies die logische
                        ' y-Koordinate des Bildpunktes.
   PSET (WLinks, LogikY)  ' Zeichne den äußerst linken
                          ' Bildpunkt einer Zeile.
   AltFarbe = 0     ' Beginne mit der Hintergrundfarbe.

   FOR X = 0 TO XLaenge    ' Durchlaufe jeden Bildpunkt
                           ' einer Zeile.
      LogikX = PMAP(X, 2)  ' Lies die logische
                           ' x-Koordinate des
                           ' Bildpunktes.
      MandelX& = LogikX
      MandelY& = LogikY

      ' Führe die Berechnungen aus, um festzustellen,
      ' ob dieser Punkt sich in der Mandelbrot-Menge
      ' befindet.

      FOR I = 1 TO MAXSCHLEIFE
         RealZahl& = MandelX& * MandelX&
         ImagZahl& = MandelY& * MandelY&
         IF (RealZahl& +ImagZahl&) >= MAXGROSS THEN EXIT FOR
         MandelY& = (MandelX& * MandelY&) \ 250 + LogikY
         MandelX& = (RealZahl& - ImagZahl&) \ 500 + LogikX
      NEXT I

      ' Weise dem Punkt eine Farbe zu:
      PFarbe = I \ FarbeBreit

      ' Wenn die Farbe gewechselt hat, zeichne eine
      ' Zeile von dem letzten Punkt, auf den Bezug
      ' genommen wurde, zu dem neuen Punkt und
      ' verwende die alte Farbe.
```

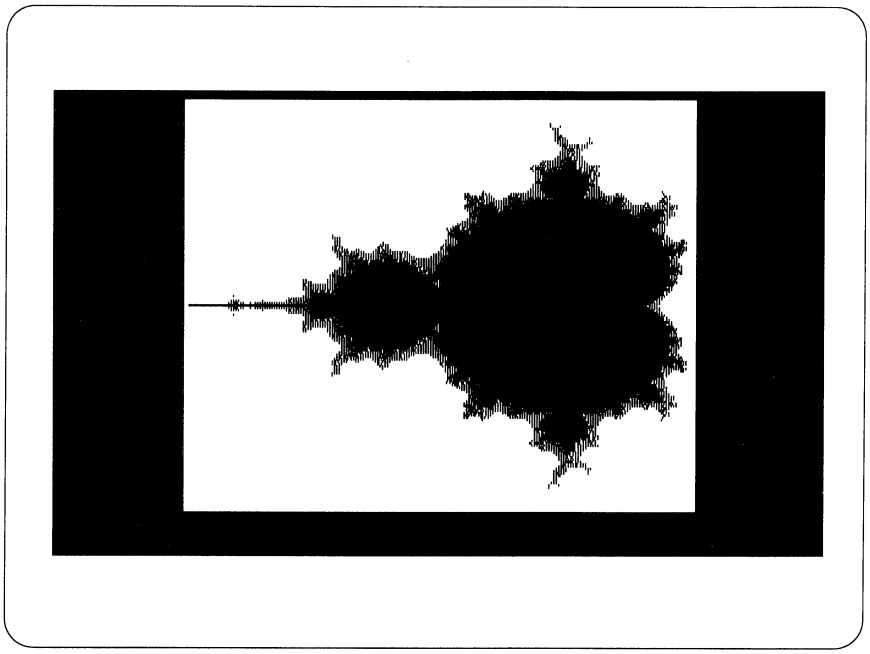
```
      IF PFarbe  <> AltFarbe THEN
        LINE -(LogikX, LogikY), (FarbeBereich - AltFarbe)
        AltFarbe = PFarbe
      END IF

      IF INKEY$ <> "" THEN END
   NEXT X

   ' Zeichne den letzten Zeilenabschnitt zur rechten
   ' Seite des Darstellungsfeldes:
   LINE -(LogikX, LogikY), (FarbeBereich - AltFarbe)

   ' Wenn es sich um eine EGA-Maschine handelt, ändere
   ' die Palette nach dem Zeichnen jeder Zeile:
   IF EgaModus THEN AenderPalette
NEXT Y

DO
   ' Verändere die Palette solange, bis der Benutzer
   ' eine Taste betätigt:
   IF EgaModus THEN AenderPalette
LOOP WHILE INKEY$ = ""

SCREEN 0, 0        ' Schalte den Bildschirm wieder auf
WIDTH 80           ' den Textmodus, 80 Spalten.
END

FalschMonit:           ' Fehlerbehandlung, die aufgeru-
   EgaModus = FALSCH   ' fen wird, wenn keine EGA-
   RESUME NEXT         ' Grafikkarte vorhanden ist.
'
' ================= AenderPalette ====================
'   Verschiebt die Palette bei jedem Aufruf um eins
' ====================================================
'
SUB AenderPalette STATIC
   SHARED PaletteFeld(), FarbeBereich

   FOR I = 1 TO FarbeBereich
      PaletteFeld(I) = (PaletteFeld(I) MOD FarbeBereich) + 1
   NEXT I
   PALETTE USING PaletteFeld(0)

END SUB
'
' ================== MonitorTest ===================
' Verwendet eine Anweisung SCREEN 8 als Prüfung, ob
' der Benutzer EGA-Hardware einsetzt. Wenn sich dabei
' ein Fehler ergibt, wird die EM-Flagge auf FALSCH
' gesetzt, und der Bildschirm wird mit SCREEN 1
' gesetzt.
'
```

```
' Setzt ebenso Werte für die Ecken des
' Darstellungsfeldes (DL = Links, DR = Rechts,
' DOB = Oben, DU = Unten), skaliert mit dem
' richtigen Bildverhältnis, so daß das
' Darstellungsfeld wirklich ein Quadrat ist.
' ======================================================
'
SUB MonitorTest (EM, FB, DL, DR, DOB, DU) STATIC
   EM = WAHR
   ON ERROR GOTO FalschMonit
   SCREEN 8, 1
   ON ERROR GOTO 0
   IF EM THEN      ' Kein Fehler, also ist SCREEN 8 OK.
      DL = 110 : DR = 529
      DOB = 5 : DU = 179
      FB = 15      ' 16 Farben (0 - 15)
   ELSE            ' Fehler, also verwende SCREEN 1.
      SCREEN 1, 1
      DL = 55 : DR = 264
      DOB = 5 : DU = 179
      FB = 3       ' 4 Farben (0 - 3)
   END IF
END SUB
'
' ================== FenstWerte ====================
' Liest Fensterecken als Benutzereingabe oder setzt
' die Werte der Ecken, wenn keine Eingabe vorhanden
' ist.
' ======================================================
'
SUB FenstWerte (FL, FR, FO, FU) STATIC
   CLS
   PRINT "Dieses Programm druckt die grafische ";
   PRINT "Darstellung der vollständigen Mandelbrot-Menge."
   PRINT "Das Standardfenster reicht von (-1000,625) bis";
   PRINT " (250,-625)."
   PRINT " Zum Vergrößern eines Teils der Figur geben Sie";
   PRINT " Koordinaten innerhalb dieses Fensters ein."
   PRINT
```

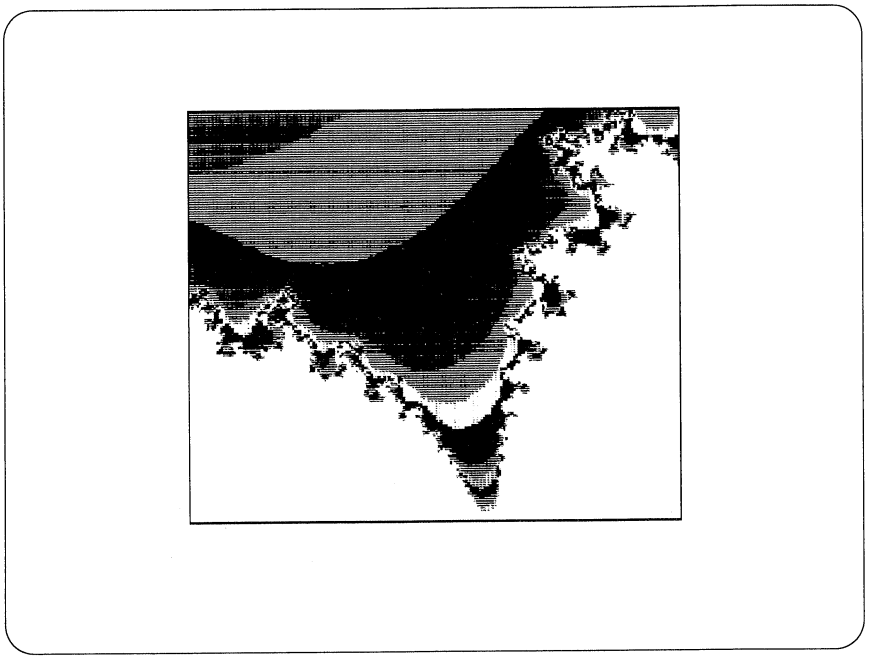
```
    PRINT "Nach Betätigen von <EINGABETASTE> sehen Sie ";
    PRINT " das Standardfenster."
    PRINT "Betätigen Sie eine andere Taste, um eigene ";
    PRINT "Fensterkoordinaten einzugeben:";
    LOCATE , , 1
    Antw$ = INPUT$(1)
    ' Benutzer hat EINGABETASTE nicht betätigt, also
    ' lies Fensterecken ein:
    IF Antw$ <> CHR$(13) THEN
       PRINT
       INPUT "X-Koordinate der oberen linken Ecke: ", FL
       DO
          INPUT "X-Koordinate der unteren rechten Ecke: ", FR
          IF FR <= FL THEN
             PRINT "Rechte Ecke muß größer als linke Ecke_
                   sein."
          END IF
       LOOP WHILE FR <= FL
       INPUT "Y-Koordinate der oberen linken Ecke: ", FO
       DO
          INPUT "Y-Koordinate der unteren rechten Ecke: ", FU
          IF FU >= FO THEN
             PRINT "Untere Ecke muß kleiner als obere Ecke_
                   sein."
          END IF
       LOOP WHILE FU >= FO
    ' Benutzer betätigte EINGABETASTE, also setze
    ' Standardwerte:
    ELSE
       FL = -1000
       FR =   250
       FO =   625
       FU =  -625
    END IF
END SUB
```

### *Ausgabe*

Die folgende Abbildung zeigt die vollständige Mandelbrot-Figur, wie sie im Bildschirmmodus 1 erscheint. Dies ist die Ausgabe, die Sie sehen, wenn Sie eine CGA haben und die Koordinaten des Standardfensters wählen.



Die nächste Abbildung zeigt die Mandelbrot-Figur mit (*x*,*y*)-Koordinaten von (`-500, 250`) für die obere linke Ecke und (`-300, 50`) für die untere rechte Ecke. Diese Abbildung ist im Bildschirmmodus 8 gezeichnet, den das Programm wählt, wenn Sie eine EGA oder VGA haben.



## 5.11.3 Editor für Muster (EDMUST.BAS)

Dieses Programm ermöglicht es Ihnen, eine Musterkachel für die Verwendung mit **PAINT** zu editieren. Während Sie die Kachel auf der linken Seite des Bildschirms editieren, können Sie das Erscheinungsbild des fertigen Musters auf der rechten Seite des Bildschirms sehen. Wenn Sie das Editieren der Musterkachel beendet haben, gibt das Programm die ganzzahligen Argumente aus, die die Anweisung **CHR$** zum Zeichnen jeder Zeile der Kachel verwendet.

### Verwendete Anweisungen und Funktionen

Dieses Programm demonstriert die Verwendung der folgenden Grafikanweisungen:

- **LINE**
- **PAINT** (mit Muster)
- **VIEW**

**Programm-Listing**

```
DECLARE SUB ZeichneMuster ()
DECLARE SUB EditMuster    ()
DECLARE SUB Initialisiere ()
DECLARE SUB ZeigMuster    (OK$)

DIM Bit%(0 TO 7), Muster$, Esc$, MusterGross%

DO
   Initialisiere
   EditMuster
   ZeigMuster OK$
LOOP WHILE OK$ = "J"

END
'
' ================== ZEICHNEMUSTER ==================
'    Zeichnet ein gemustertes Rechteck auf der
'    rechten Seite des Bildschirms.
' ===================================================
'
SUB ZeichneMuster STATIC
SHARED Muster$

   VIEW (320, 24)-(622, 160), 0, 1 ' Setze Feld auf
                                   ' Rechteck.
   PAINT (1, 1), Muster$      ' Verwende PAINT, um es
                              ' auszufüllen.
   VIEW                       ' Setze Feld auf gesamten
                              ' Bildschirm.

END SUB
'
' ================== EDITMUSTER ====================
'       Editiert ein Kachel-Bytemuster
'==================================================
'
SUB EditMuster STATIC
SHARED Muster$, Esc$, Bit%(), MusterGross%

   ByteZahl% = 1     ' Anfangsposition.
   BitZahl% = 7
   Null$ = CHR$(0)   ' CHR$(0) ist das erste zurückgegebene
                     ' Byte der 2-Byte-Zeichenkette, wenn
                     ' eine Richtungstaste wie OBEN oder
                     ' UNTEN betätigt wurde.
```

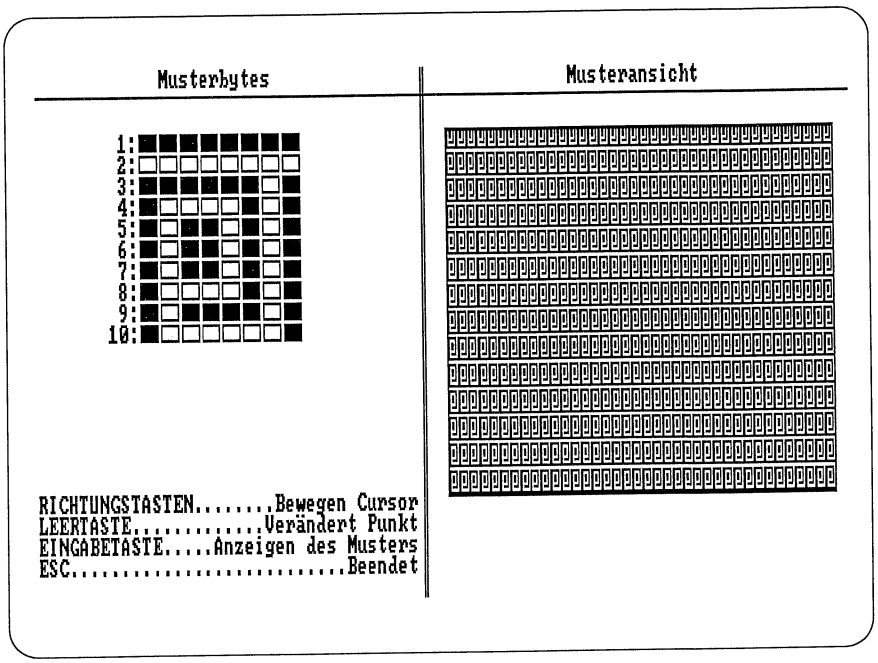
```
DO
   ' Berechne Anfangsposition dieses Bits auf dem
   ' Bildschirm:
   X% = ((7 - BitZahl%) * 16) + 80
   Y% = (ByteZahl% + 2) * 8
   ' Warte auf Tastenbetätigung (und lasse Cursor
   ' alle 3/10 Sekunden blinken):
   Zustand% = 0
   AufZeit = 0
   DO
   ' Prüfe Zeitgeber und wechsle Cursor-Zustand,
   ' wenn 3/10 Sekunden vergangen sind:
   IF ABS(TIMER - AufZeit) > .3 THEN
      AufZeit = TIMER
      Zustand% = 1 - Zustand%
      ' Schalte Rahmen des Bits an und aus:
      LINE (X%-1, Y%-1)-STEP(15, 8), Zustand%, B
   END IF
   Pruef$ = INKEY$    ' Prüfe auf Tastenbetätigung.
   LOOP WHILE Pruef$ = "" ' Durchlaufe Schleife,
                          ' bis eine Taste betätigt
                          ' ist.
   ' Lösche Cursor:
   LINE (X%-1, Y%-1)-STEP(15, 8), 0, B
   SELECT CASE Pruef$   ' Reagiere auf
                        ' Tastenbetätigung.
      CASE CHR$(27)     ' ESC-Taste betätigt:
         EXIT SUB       ' verlasse dieses
                        ' Unterprogramm.
      CASE CHR$(32)     ' LEERTASTE betätigt:
                        ' setze Zustand des Bits
                        ' neu.
         ' Invertiere Bit in der
         ' Musterzeichenkette:
         AktuellByte% = ASC(MID$(Muster$, ByteZahl%, 1))
         AktuellByte% = AktuellByte% XOR Bit%(BitZahl%)
         MID$ (Muster$, ByteZahl%) = CHR$(AktuellByte%)
```

```
            ' Zeichne Bit neu auf den Bildschirm:
            IF (AktuellByte% AND Bit%(BitZahl%)) <> 0 THEN
               AktuellFarbe% = 1
            ELSE
               AktuellFarbe% = 0
            END IF
            LINE (X%+1, Y%+1)-STEP(11, 4), AktuellFarbe%, BF
         CASE CHR$(13)      ' EINGABETASTE betätigt:
            ZeichneMuster   ' zeichne Muster im rechten
                            ' Rechteck.
         CASE Null$ + CHR$(75)  ' LINKS-Taste: bewege
                                ' Cursor nach links.
            BitZahl% = BitZahl% + 1
            IF BitZahl% > 7 THEN BitZahl% = 0
         CASE Null$ + CHR$(77)  ' RECHTS-Taste: bewege
                                ' Cursor nach rechts.
            BitZahl% = BitZahl% - 1
            IF BitZahl% < 0 THEN BitZahl% = 7
         CASE Null$ + CHR$(72)  ' OBEN-Taste: bewege
                                ' Cursor nach oben.
            ByteZahl% = ByteZahl% - 1
            IF ByteZahl% < 1 THEN
               ByteZahl% = MusterGross%
            END IF
         CASE Null$ + CHR$(80)  ' UNTEN-Taste: bewege
                                ' Cursor nach unten.
            ByteZahl% = ByteZahl% + 1
            IF ByteZahl% > MusterGross% THEN ByteZahl% = 1
         CASE ELSE
            ' Benutzer hat eine andere Taste als ESC,
            ' LEERTASTE, EINGABETASTE, OBEN, UNTEN,
            ' LINKS oder RECHTS betätigt, also tue
            ' nichts.
      END SELECT
   LOOP
END SUB
```

```
'
' ================== INITIALISIERE ==================
'  Initialisiert Anfangsmuster und Bildschirm.
' ===================================================
'
SUB Initialisiere STATIC
SHARED Muster$, Esc$, Bit%(), MusterGross%
   Esc$ = CHR$(27)     ' ESC-Zeichen ist ASCII 27.
   ' Initialisiere ein Datenfeld, das Bits in den
   ' Positionen 0 bis 7 enthält:
   FOR I% = 0 TO 7
      Bit%(I%) = 2 ^ I%
   NEXT I%
   CLS
   ' Eingabe der Größe des Musters (in Anzahl an
   ' Bytes):
   LOCATE 5, 5
   PRINT "Geben Sie die Größe des Musters ein ";
   PRINT "(1-16 Zeilen):";
   DO
      LOCATE 5, 58
      PRINT "    ";
      LOCATE 5, 58
      INPUT "", MusterGross%
   LOOP WHILE MusterGross% < 1 OR MusterGross% > 16
   ' Setze Anfangsmuster auf alle Bits gesetzt:
   Muster$ = STRING$(MusterGross%, 255)
   SCREEN 2     ' 640 x 200 monochromer Grafikmodus
   ' Zeichne Trennungslinien:
   LINE (0, 10)-(635, 10),1
   LINE (300, 0)-(300, 199)
   LINE (302, 0)-(302, 199)
   ' Schreibe Überschriften:
   LOCATE 1, 13: PRINT "Musterbytes"
   LOCATE 1, 53: PRINT "Musteransicht"
   ' Zeichne Editierbildschirm für das Muster:
   FOR I% = 1 TO MusterGross%
      ' Gib links von jeder Zeile Bezeichnung aus:
      LOCATE I% + 3, 8
      PRINT USING "##:"; I%
```

```
      ' Zeichne "Bit"-Rechtecke:
      X% = 80
      Y% = (I% + 2) * 8
      FOR J% = 1 TO 8
        LINE (X%, Y%)-STEP(13, 6), 1, BF
        X% = X% + 16
      NEXT J%
   NEXT I%

   ZeichneMuster   ' Zeichne "Musteransicht"-Rechteck.

   LOCATE 21, 1
   PRINT "RICHTUNGSTASTEN........Bewegen Cursor"
   PRINT "LEERTASTE.............Verändert Punkt"
   PRINT "EINGABETASTE.....Anzeigen des Musters"
   PRINT "ESC..........................Beendet";
END SUB
'
' ==================== ZEIGMUSTER ====================
'     Gibt die von PAINT zum Erstellen des Musters
'     verwendeten CHR$-Werte
' ====================================================
'
SUB ZeigMuster (OK$) STATIC
SHARED Muster$, MusterGross%

   ' Schalte Bildschirm zurück in 80-Spalten Textmodus:
   SCREEN 0, 0
   WIDTH 80

   PRINT "Die folgenden Zeichen erstellen Ihr Muster:"
   PRINT

   ' Gib den Wert für jedes Musterbyte aus:
   FOR I% = 1 TO MusterGross%
      MusterByte% = ASC(MID$(Muster$, I%, 1))
      PRINT "CHR$("; LTRIM$(STR$(MusterByte%)); ")"
   NEXT I%

   PRINT
   LOCATE ,,1
   PRINT "Neues Muster? ";
   OK$ = UCASE$(INPUT$(1))
END SUB
```

**Ausgabe**




## 5.12 Zusammenfassung der Grafikanweisungen und -funktionen

Die folgende Tabelle führt alle Anweisungen und Funktionen auf, die in BASIC für auf Bildpunkte aufbauende Grafiken verwendet werden.

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Setzen der Eigenschaften des Bildschirms | **SCREEN** | Legt einen BASIC-Bildschirmmodus fest, der Bildschirmeigenschaften, wie Auflösung und Bereiche für Farbnummern, bestimmt |
| Zeichnen oder Löschen eines einzelnen Punktes | **PSET** | Gibt einem Bildpunkt auf dem Bildschirm eine bestimmte Farbe; verwendet standardmäßig die Vordergrundfarbe des Bildschirms |

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Zeichnen oder Löschen eines einzelnen Punktes | **PRESET** | Gibt einem Bildpunkt auf dem Bildschirm eine bestimmte Farbe; verwendet standardmäßig die Hintergrundfarbe des Bildschirms, löscht damit effektiv den Bildpunkt |
| Zeichnen einfacher Figuren auf dem Bildschirm | **LINE** | Zeichnet eine gerade Linie oder ein Rechteck |
| | **CIRCLE** | Zeichnet einen Kreis oder eine Ellipse |
| | **DRAW** | Kombiniert viele der Eigenschaften anderer BASIC-Grafikanweisungen (Zeichnen von Geraden, Bewegen des grafischen Cursors, Skalieren von Bildern) in einer "all-in-one" grafischen Makro-Sprache |
| Definieren von Bildschirmkoordinaten | **VIEW** | Legt ein Rechteck auf dem Bildschirm (oder Darstellungsfeld) als den Bereich für grafische Ausgabe fest |
| | **WINDOW** | Ermöglicht es dem Benutzer, neue logische Koordinaten für ein Darstellungsfeld auf dem Bildschirm zu wählen |
| | **PMAP** | Bildet physikalische Bildschirmkoordinaten auf vom Benutzer im aktuellen Fenster festgelegte logische Koordinaten ab bzw. umgekehrt |
| | **POINT**(*Zahl*) | Gibt die aktuellen physikalischen oder logischen Koordinaten des grafischen Cursors, abhängig vom Wert für *Zahl*, an |
| Verwenden von Farben | **COLOR** | Setzt die in grafischer Ausgabe standardmäßig verwendeten Farben |
| | **PALETTE** | Weist Farbnummern andere Farben zu. Funktioniert nur auf Systemen, die mit einer EGA oder VGA ausgerüstet sind |

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Verwenden von Farben | **POINT**(*x*,*y*) | Gibt die Farbnummer eines einzelnen Bildpunktes an, dessen Bildschirmkoordinaten *x* und *y* sind |
| Ausmalen geschlossener Formen | **PAINT** | Füllt einen Bildschirmbereich mit einer Farbe oder einem Muster |
| Animation | **GET** | Kopiert einen rechteckigen Bereich des Bildschirms durch Übertragen des Bildes in numerische Daten und Speichern dieser Daten in einem numerischen Datenfeld |
| | **PUT** | Zeigt ein Bild auf dem Bildschirm an, das zuvor mit **GET** kopiert wurde |
| | **PCOPY** | Kopiert eine Bildschirmseite in eine andere |

# 6 Fehler- und Ereignisverfolgung


6.1 Fehlerverfolgung 6.2
- 6.1.1 Wie Sie Fehlerverfolgung aktivieren 6.3
- 6.1.2 Wie Sie eine Fehlerbehandlungsroutine schreiben 6.4
  - 6.1.2.1 Wie Sie **ERR** zum Identifizieren von Fehlern einsetzen 6.4
  - 6.1.2.2 Verlassen einer Fehlerbehandlungsroutine 6.6

6.2 Ereignisverfolgung 6.9
- 6.2.1 Wie Sie Ereignisse erfassen: Registrierung und Verfolgung 6.9
  - 6.2.1.1 Registrierung 6.9
  - 6.2.1.2 Verfolgung 6.9
- 6.2.2 Wie Sie das zu verfolgende Ereignis angeben und Ereignisverfolgung aktivieren 6.10
- 6.2.3 Ereignisse, die BASIC verfolgen kann 6.11
- 6.2.4 Wie Sie Ereignisverfolgung aussetzen oder ausschalten 6.12
- 6.2.5 Wie Sie Tastenbetätigungen verfolgen 6.13
  - 6.2.5.1 Wie Sie eine beliebige benutzerdefinierte Taste verfolgen 6.15
  - 6.2.5.2 Wie Sie benutzerdefinierte umgeschaltete Tasten verfolgen 6.15
- 6.2.6 Wie Sie Musikereignisse verfolgen 6.18

6.3 Fehler- und Ereignisverfolgung in **SUB**- oder **FUNCTION**-Prozeduren 6.21

6.4 Verfolgung in mehrmoduligen Programmen 6.22
- 6.4.1 Ereignisverfolgung quer durch Module 6.22
- 6.4.2 Fehlerverfolgung in mehrmoduligen Programmen 6.23

6.5 Fehler- und Ereignisverfolgung in Programmen, die mit bc kompiliert sind 6.26

6.6 Beispielanwendung: Verfolgen von Dateizugriffsfehlern (DATFEHL.BAS) 6.28

6.7 Zusammenfassung der Anweisungen und Funktionen zur Verfolgung 6.31

Dieses Kapitel zeigt Ihnen, wie Fehler und Ereignisse verfolgt werden, die während der Programmausführung auftreten. Mit Fehlerverfolgung können Sie Ihr Programm vor Benutzerfehlern schützen, wie Eingabe einer Zeichenkette wenn ein numerischer Wert erwartet wird, oder einer Anforderung, eine Datei in einem nicht existierenden Verzeichnis zu öffnen. Mit Ereignisverfolgung versetzen Sie Ihr Programm in die Lage, Laufzeitereignisse zu entdecken und zu verarbeiten, wie Tastenbetätigungen oder einen Wechsel in der Anzahl an Noten, die zum Spielen in einer Folge für Hintergrundmusik übriggeblieben sind.

Wenn Sie dieses Kapitel beendet haben, werden Sie wissen, wie die folgenden Aufgaben durchgeführt werden:

- Einschalten von Fehler- oder Ereignisverfolgung.
- Schreiben einer Routine, um verfolgte Fehler oder Ereignisse zu verarbeiten.
- Zurückgeben der Kontrolle von einer Fehlerbehandlungsroutine oder einer Ereignisverfolgungsroutine.
- Schreiben eines Programmes, das jede Tastenbetätigung bzw. Kombination von Tastenbetätigungen verfolgt.
- Verfolgen von Fehlern oder Ereignissen innerhalb von **SUB**- oder **FUNCTION**-Prozeduren.
- Verfolgen von Fehlern und Ereignissen in Programmen, die aus mehr als einem Modul zusammengesetzt sind.

## 6.1 Fehlerverfolgung

Die Fehlerverfolgung erlaubt Ihnen Programme zu schreiben, die Laufzeitfehler abfangen. Wenn BASIC auf einen Fehler trifft, wie zum Beispiel den Versuch, während der Ausführung eines Programmes eine nichtexistierende Datendatei zu öffnen, zeigt es ohne Fehlerverfolgung seine eigene Fehlermeldung an und beendet dann das Programm. An diesem Punkt wird der Benutzer in die QuickBASIC-Entwicklungsumgebung oder, wenn es sich um ein kompiliertes Programm handelt, auf Betriebssystemebene zurückversetzt. Wenn der Benutzer eine kompilierte, selbständige Version Ihres Programmes verwendet, besteht die Möglichkeit, daß er oder sie das Programm neu starten muß, wodurch manchmal Daten oder Berechnungen, die das Programm vor Auftreten des Fehlers durchgeführt hatte, verloren gehen.

Mit Fehlerverfolgung jedoch beenden Laufzeitfehler das Programm nicht. Stattdessen veranlassen diese eine Verzweigung zu einer "Fehlerbehandlungsroutine", die die notwendigen Schritte zur Korrektur des Fehlers vornehmen kann, wie zum Beispiel die Ausgabe eigener benutzerspezifischer Fehlermeldungen.

Fehlerverfolgung erlaubt es Ihnen daher, bestimmte Fehler abzufangen, die auftreten können, wenn ein anderer Benutzer Ihr Programm verwendet, und mit solchen Fehlern in einer "benutzerfreundlichen" Art umzugehen.

Die Abschnitte 6.1.1 und 6.1.2 zeigen, wie Fehlerverfolgung aktiviert wird, wie eine Routine zur Behandlung von Fehlern, wenn diese verfolgt werden, geschrieben wird, und wie die Kontrolle von der Fehlerbehandlungsroutine zurückgegeben wird, nachdem diese den Fehler bearbeitet hat.

### 6.1.1 Wie Sie Fehlerverfolgung aktivieren

Innerhalb eines Moduls wird Fehlerverfolgung mit der Anweisung

**ON ERROR GOTO** *Zeile*

aktiviert, wobei *Zeile* eine gültige Zeilennummer oder Zeilenmarke ist, die die erste Zeile einer Fehlerbehandlungsroutine kennzeichnet. Sobald BASIC eine Anweisung **ON ERROR GOTO** *Zeile* vorgefunden hat, verursacht jeder Laufzeitfehler innerhalb dieses Moduls eine Verzweigung in *Zeile*.

***Hinweis*** Sie können eine Anweisung **ON ERROR GOTO 0** nicht dazu verwenden, in eine Fehlerbehandlung zu verzweigen, da diese Anweisung für Fehlerverfolgung eine spezielle Bedeutung hat. Die Anweisung **ON ERROR GOTO 0** hat abhängig davon, wo sie erscheint, zwei Auswirkungen. Wenn sie außerhalb einer Fehlerbehandlungsroutine erscheint, schaltet sie Fehlerverfolgung aus; das heißt, nachdem das Programm auf ein **ON ERROR GOTO 0** gestoßen ist, beendet jeder nachfolgende Fehler das Programm, anstatt die Fehlerbehandlungsroutine aufzurufen. Wenn die Anweisung innerhalb einer Fehlerbehandlungsroutine erscheint, wie dies in der Routine `Behandlung` im nächsten Beispiel geschieht, veranlaßt diese Anweisung BASIC dazu, dessen Standardmeldung für den vorliegenden Fehler auszugeben, und beendet das Programm.

Selbst dann, wenn Ihr Programm eine Zeile mit der Zeilennummer 0 hat, signalisiert eine Anweisung **ON ERROR GOTO 0** daher dem Programm, Fehlerfeststellung auszuschalten oder die Ausführung zu beenden.

## 6.1.2 Wie Sie eine Fehlerbehandlungsroutine schreiben

Eine Fehlerbehandlungsroutine besteht aus den folgenden drei Teilen:

1. Die in der Anweisung **ON ERROR GOTO** *Zeile* angegeben Zeilenmarke oder Zeilennummer, die die erste Anweisung darstellt, zu der das Programm nach einem Fehler verzweigt.
2. Dem Hauptteil der Routine, der den die Verzweigung verursachenden Fehler identifiziert und auf jeden abgefangenen Fehler entsprechend reagiert.
3. Einer oder mehreren **RESUME**- oder **RESUME NEXT**-Anweisungen, um die Kontrolle aus dieser Routine zurückzugeben.

Eine Fehlerbehandlungsroutine kann nicht innerhalb einer **SUB**- oder **FUNCTION**-Prozedur oder innerhalb einer **DEF FN**-Funktion stehen. Außerdem sollte sie so plaziert werden, daß der normale Ablauf der Ausführung nicht durch diese Routine führt; wenn sich die Routine zum Beispiel im Hauptmodul eines mehrmoduligen Programmes befindet, lassen Sie ihr eine **END**-Anweisung vorangehen.

### 6.1.2.1 Wie Sie ERR zum Identifizieren von Fehlern einsetzen

Sobald ein Programm in eine Fehlerbehandlungsroutine verzweigt hat, benötigt es eine Möglichkeit, die unterschiedlichen Fehler zu trennen, die die Verzweigung verursacht haben könnten. Verwenden Sie die Funktion **ERR**, um die spezielle Fehlerquelle zu identifizieren. Diese Funktion gibt einen numerischen Code für den letzten Laufzeitfehler des Programmes an. (Eine komplette Liste der Fehlercodes, die mit Laufzeitfehlern verknüpft sind, finden Sie im Anhang D, "Fehlermeldungen", im *BASIC-Befehlsverzeichnis*.)

***Hinweis*** Fehler können nicht innerhalb einer Fehlerbehandlungsroutine verfolgt werden. Wenn ein Fehler auftritt, während die Fehlerbehandlungsroutine einen anderen Fehler verarbeitet, gibt das Programm die Meldung für den neuen Fehler aus und beendet dann die Ausführung.

#### Beispiel

Das folgende Programm hat eine als `Behandlung` benannte Fehlerbehandlungsroutine, die dazu entworfen ist, speziell mit drei unterschiedlichen Fehlersituationen umzugehen. Die in einer **SELECT CASE**-Anweisung verwendete **ERR**-Funktion ermöglicht es dieser Routine, auf jeden Fehler passend zu reagieren.

```
' Daten für dieses Programm:
DATA BASIC, Pascal, FORTRAN, C, Modula, Forth
DATA LISP, ADA, COBOL
' Definiere symbolische Konstanten:
CONST FALSCH = 0, WAHR = NOT FALSCH
EndeDaten = FALSCH

ON ERROR GOTO Behandlung        ' Aktiviere
                                ' Fehlerverfolgung.
OPEN "LPT1:" FOR OUTPUT AS #1   ' Öffne den Drucker für
                                ' Ausgabe.
DO
   ' Fahre mit dem Lesen von Größen aus den obigen
   ' DATA-Anweisungen fort, gib sie dann auf den
   ' Zeilendrucker aus, bis eine Fehlermeldung
   ' "Ende der Eingabedaten" anzeigt, daß keine weiteren
   ' Daten vorhanden sind:
   READ Puffer$
   IF NOT EndeDaten THEN
      PRINT #1, Puffer$
   ELSE
      EXIT DO
   END IF
LOOP
CLOSE #1
END
Behandlung:     ' Fehlerbehandlungsroutine
         ' Verwende ERR um zu bestimmen, welcher Fehler die
         ' Verzweigung in "Behandlung" verursachte:
   SELECT CASE ERR
      CASE 4
         ' 4 ist der Code für "Außerhalb von DATA" in
         ' DATA-Anweisungen:
         EndeDaten = WAHR
         RESUME NEXT
      CASE 25
         ' 25 ist der Code für "Gerätefehler", der
         ' durch den Versuch verursacht sein kann,
         ' Ausgabe auf den Drucker zu geben, wenn
         ' dieser nicht eingeschaltet ist:
```

```
            PRINT "Schalten Sie den Drucker ein, betätigen ";
            PRINT "Sie dann eine beliebige Taste zum ";
            PRINT "Fortfahren."
            Pause$ = INPUT$(1)
            RESUME
        CASE 27
            ' 27 ist der Code für "Papier zu Ende":
            PRINT "Drucker hat kein Papier mehr. Legen";
            PRINT " Sie Papier ein,betätigen Sie dann eine";
            PRINT " beliebige Taste zum Fortfahren."
            Pause$ = INPUT$(1)
            ' Beginne, Daten vom Anfang der ersten DATA-
            ' Anweisung zu lesen, nachdem das Papier
            ' eingelegt ist:
            RESTORE
            RESUME
        CASE ELSE
            ' Ein nicht vom Programmierer abgefangener
            ' Fehler trat auf; zeige die Fehlermeldung
            ' für diesen Fehler an und beende das
            ' Programm:
            ON ERROR GOTO 0
    END SELECT
```

### 6.1.2.2 Verlassen einer Fehlerbehandlungsroutine

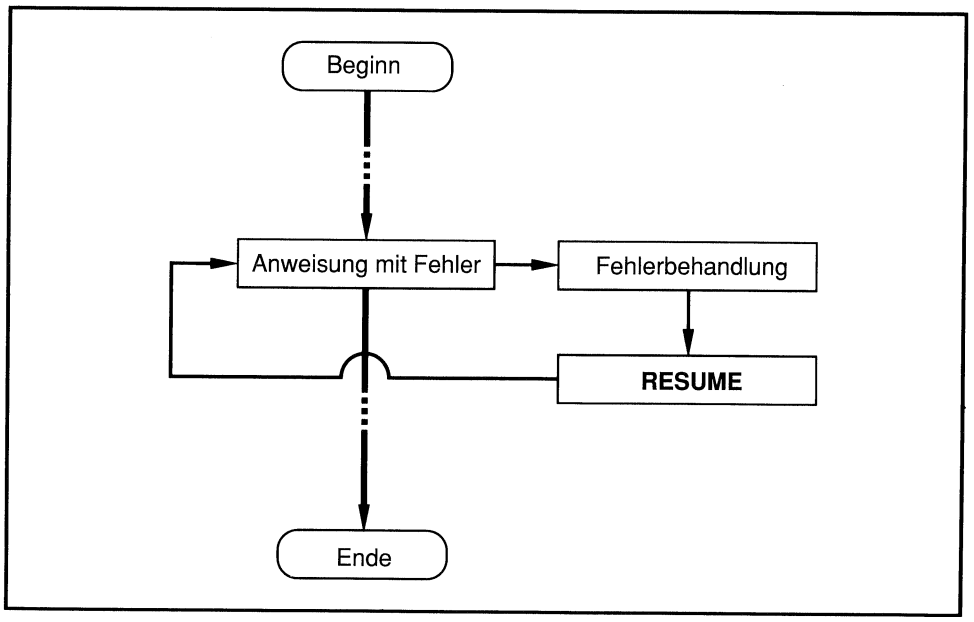
Die Anweisung **RESUME** gibt die Kontrolle von einer Fehlerbehandlungsroutine zurück. Das vorhergehende Beispiel verwendet die folgenden zwei Abwandlungen von **RESUME**, die Routine `Behandlung` zu verlassen:

| *Anweisung* | *Funktion* |
|---|---|
| **RESUME** | Veranlaßt das Programm, genau zu der Anweisung zurückzukehren, die den Fehler verursachte.<br>Das vorhergehende Beispiel verwendet **RESUME**, um zu derjenigen **PRINT**-Anweisung zurückzukehren, die versucht hat, Ausgabe auf den Drucker zu senden, und gibt der **PRINT**-Anweisung erneut die Möglichkeit, den Wert in `Puffer$` auszudrucken, nachdem der Drucker vermutlich eingeschaltet ist. |

| *Anweisung* | *Funktion* |
|---|---|
| **RESUME NEXT** | Veranlaßt das Programm, zu der Anweisung zurückzukehren, die derjenigen folgt, die den Fehler verursachte.<br><br>Das vorhergehende Programm verwendet die Anweisung **RESUME NEXT** zum Abfangen des Fehlers `Außerhalb von DATA`. Ein einfaches **RESUME** wäre in diesem Fall ein Fehler: Es würde das Programm in eine Endlosschleife schicken, da jedesmal, wenn die Kontrolle zu der **READ**-Anweisung im Hauptprogramm zurückkehrt, ein weiterer Fehler `Außerhalb von DATA` erscheint, der die Fehlerroutine erneut aufruft usw. |

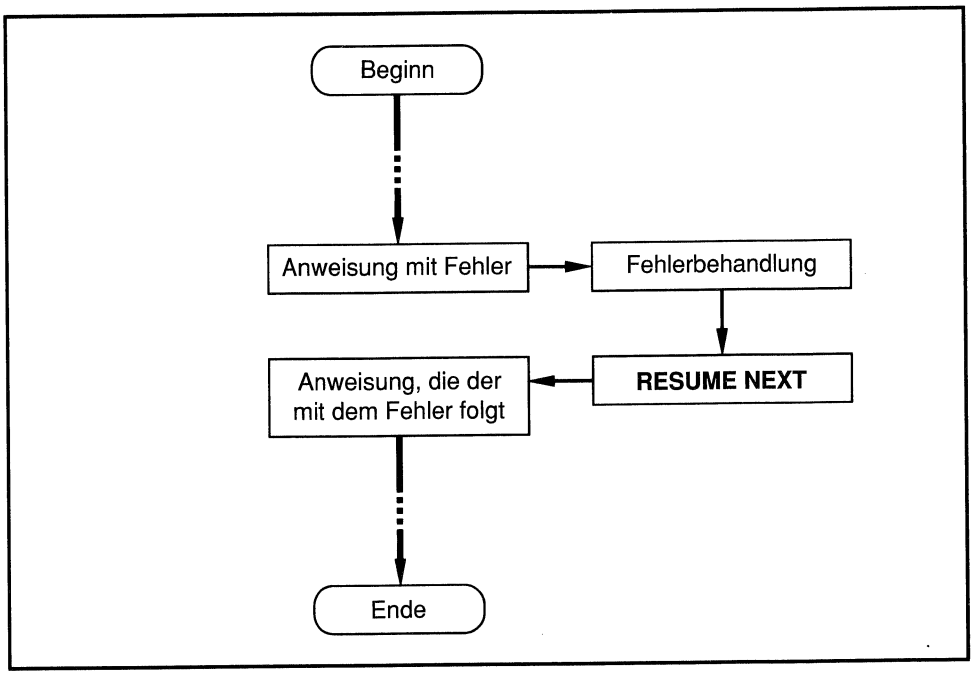
Abbildung 6.1 skizziert den Ablauf einer Programmsteuerung während einer Fehlerbehandlung mit **RESUME**:

*Abbildung 6.1 Ablauf der Programmsteuerung mit RESUME*




Vergleichen Sie die vorhergehende Abbildung mit Abbildung 6.2, die zeigt, was in einem Programm passiert, das **RESUME NEXT** verwendet:

*Abbildung 6.2 Ablauf der Programmsteuerung mit RESUME NEXT*




Es gibt eine weitere Variante von **RESUME**: Die Anweisung **RESUME** *Zeile*, wobei *Zeile* eine gültige Zeilennummer oder Zeilenmarke außerhalb irgendeines **SUB...END SUB**-, **FUNCTION...END FUNCTION**- oder **DEF FN...END DEF**-Blockes sein muß. Wegen dieser Einschränkung der Position von *Zeile* kann eine **RESUME** *Zeile*-Anweisung zu unerwünschten Seiteneffekten führen, wenn sie in einer Fehlerbehandlungsroutine erscheint, die Fehler innerhalb einer **SUB**- oder **FUNCTION**-Prozedur oder **DEF FN**-Funktion bearbeitet. Sie sollte gewöhnlich zugunsten von **RESUME** oder **RESUME NEXT** vermieden werden.

# 6.2 Ereignisverfolgung

Abschnitt 6.2.1 vergleicht zwei Methoden, mit denen Sie in Ihren BASIC-Programmen Ereignisse entdecken können: Registrierung und Verfolgung. Die Abschnitte 6.2.2 bis 6.2.4 erläutern dann die verschiedenen Ereignisse, die BASIC verfolgen kann, wie eine Verfolgung gesetzt und aktiviert wird, und wie eine Verfolgung ausgesetzt oder deaktiviert wird. Die Abschnitte 6.2.5 bis 6.2.6 bieten detailliertere Informationen zur Verfolgung von Tastenbetätigungen und Musikereignissen.

## 6.2.1 Wie Sie Ereignisse erfassen: Registrierung und Verfolgung

Die nächsten beiden Abschnitte beschreiben zwei Techniken, die Sie in Ihren Programmen zum Erfassen und Behandeln von Laufzeitfehlern verwenden können: Registrierung und Verfolgung.

### 6.2.1.1 Registrierung

Eine Möglichkeit, das Auftreten eines Ereignisses zu erfassen und die Programmkontrolle in eine passende Unterroutine zu leiten, bietet die Verwendung von "Ereignisregistrierung". Bei Ereignisregistrierung prüft Ihr Programm während seiner Ausführung explizit auf Ereignisse an bestimmten Punkten. Die folgenden Anweisungen veranlassen zum Beispiel das Programm, zwischen den Anweisungen **DO** und **LOOP UNTIL** zu verweilen, bis der Benutzer entweder die "n"- oder die "b"-Taste betätigt hat:

```
DO
   Test$ = UCASE$(INKEY$)
LOOP UNTIL Test$ = "N" OR Test$ = "B"
```

Registrierung ist dann hilfreich, wenn Sie vorher ganz genau wissen, wo Sie im Programmablauf auf ein Ereignis prüfen wollen.

### 6.2.1.2 Verfolgung

Nehmen Sie jedoch an, Sie möchten Ihr Programm nicht anhalten lassen, oder Sie möchten, daß es zwischen jeder einzelnen Anweisung prüft. In diesen Fällen ist Registrierung eine schwerfällige Technik zur Fehlererfassung, und Verfolgung wäre die bessere Möglichkeit.

### Beispiel

Das folgende Beispiel zeigt die Verwendung von Fehlerverfolgung:

```
' Veranlasse, daß das Programm immer dann in
' "FunktTast" verzweigt, wenn der Benutzer die
' Funktionstaste F1 betätigt:
ON KEY (1) GOSUB FunktTast
' Schalte "Verfolgung" für die F1-Tastenbetätigung
' ein. BASIC prüft nun im Hintergrund auf dieses
' Ereignis bis zum Programmende, oder bis es mit
' KEY(1) OFF ausgeschaltet wird oder mit KEY(1) STOP
' ausgesetzt wird:
KEY(1) ON
OPEN "Daten" FOR INPUT AS #1
OPEN "Daten.Aus" FOR OUTPUT AS #2
DO UNTIL EOF(1)
   LINE INPUT #1, ZeilPuffer$
   PRINT #2, ZeilPuffer$
LOOP
FunktTast:    ' Jedesmal wenn der Benutzer F1 betätigt,
              ' verzweigt das Programm hierhin.
   .
   .
   .
RETURN
```

## 6.2.2 Wie Sie das zu verfolgende Ereignis angeben und Ereignisverfolgung aktivieren

Wie Sie dem vorhergehenden Beispiel entnehmen können, sind für Ereignisverfolgung drei Schritte notwendig:

1. Definieren einer Zielunterroutine, die als "Unterroutine zur Ereignisverfolgung" bezeichnet wird.
2. Dem Programm mit einer **ON** *Ereignis* **GOSUB**-Anweisung mitteilen, in die Unterroutine zur Ereignisverfolgung zu verzweigen, wenn das Ereignis stattfindet.
3. Aktivieren der Verfolgung dieses Ereignisses mit einer *Ereignis* **ON**-Anweisung.

Ihr Programm kann solange kein gegebenes *Ereignis* verfolgen, bis es sowohl eine *Ereignis* **ON**- als auch eine **ON** *Ereignis* **GOSUB**-Anweisung angetroffen hat.

Eine Ereignisbehandlungsroutine ist eine normale BASIC **GOSUB**-Unterroutine: die Kontrolle geht zur ersten Zeile der Unterroutine immer dann, wenn das Ereignis erfaßt wird und geht mit einer **RETURN**-Anweisung zurück auf die Hauptebene des Moduls, das die Unterroutine enthält.

Beachten Sie die folgenden wichtigen Unterschiede zwischen der Schreibweise für Fehlerverfolgung und der Schreibweise für Ereignisverfolgung:

| ***Fehlerverfolgung*** | ***Ereignisverfolgung*** |
|---|---|
| Wird mit einer **ON ERROR GOTO** -Anweisung aktiviert, die auch die erste Zeile der Fehlerbehandlungsroutine angibt. | Wird mit einer *Ereignis* **ON**-Anweisung aktiviert. Eine separate **ON** *Ereignis* **GOSUB**-Anweisung gibt die erste Zeile der Unterroutine zur Ereignisbehandlung an. |
| Gibt die Kontrolle aus der Fehlerroutine mit einer oder mehreren **RESUME**-, **RESUME NEXT**- oder **RESUME** *Zeile*-Anweisungen zurück. | Gibt die Kontrolle aus der Ereignisunterroutine mit einer oder mehreren **RETURN**-Anweisungen zurück. |

## 6.2.3 Ereignisse, die BASIC verfolgen kann

Die folgende Tabelle führt die Ereignisse auf, die Sie innerhalb eines BASIC-Programmes verfolgen können:

| ***BASIC-Anweisung*** | ***Verfolgtes Ereignis*** |
|---|---|
| **COM**(*Anschlußnummer*) | Daten aus einem der seriellen Anschlüsse (1 oder 2), die im Datenübertragungspuffer erscheinen (der Zwischenspeicherbereich für Daten, die zum seriellen Anschluß gesendet oder von diesem empfangen werden). |
| **KEY**(*Tastennummer*) | Der Benutzer betätigt die angegebene Taste. |
| **PEN** | Der Benutzer aktiviert den Lichtstift. |
| **PLAY**(*Warteschlangengrenze*) | Die Anzahl der noch im Hintergrund zu spielenden Noten fällt unterhalb der *Warteschlangengrenze*. |
| **STRIG**(*Auslösenummer*) | Der Benutzer betätigt den Auslöser des Joysticks. |
| **TIMER**(*Intervall*) | Der Ablauf von *Intervall* Sekunden. |

## 6.2.4 Wie Sie Ereignisverfolgung aussetzen oder ausschalten

Sie können die Erfassung und Verfolgung jedes Ereignisses mit der folgenden Anweisung ausschalten:

*Ereignis* **OFF**

Ereignisse, die nach der Ausführung einer *Ereignis* **OFF**-Anweisung auftreten, werden ignoriert.

Wenn Sie die Verfolgung eines gegebenen Ereignisses aussetzen wollen, das Ereignis aber erfassen wollen, verwenden Sie die folgende Anweisung:

*Ereignis* **STOP**

Nach *Ereignis* **STOP** wird, wenn das Ereignis stattfindet, nicht in die Ereignisunterroutine verzweigt. Ihr Programm hält jedoch fest, daß das Ereignis stattfand; sobald Verfolgung mit *Ereignis* **ON** wieder eingeschaltet ist, wird eine sofortige Verzweigung in die Ereignisunterroutine vorgenommen.

Unterroutinen zur Ereignisbehandlung führen eine implizite *Ereignis* **STOP**-Anweisung für ein gegebenes Ereignis immer dann aus, wenn sich die Programmkontrolle in der Unterroutine befindet; dieser Anweisung folgt ein implizites *Ereignis* **ON** für dieses Ereignis, wenn die Programmkontrolle aus der Unterroutine zurückgegeben wird. Das heißt, wenn eine Unterroutine zur Tastenbehandlung eine Tastenbetätigung bearbeitet, ist die Verfolgung dieser Taste ausgesetzt, bis die vorhergehende Tastenbetätigung von der Unterroutine vollständig bearbeitet ist. Wenn der Benutzer diese Taste währenddessen betätigt, wird diese neue Tastenbetätigung festgehalten und verfolgt, nachdem die Kontrolle aus der Unterroutine zur Tastenbehandlung zurückgegeben wird.

### Beispiel

Eine Ereignisverfolgung unterbricht alles, was zu dem Zeitpunkt, zu dem das Ereignis auftritt, in dem Programm passiert. Wenn Sie einen Block von Anweisungen haben der nicht unterbrochen werden soll, stellen Sie den Anweisungen ein *Ereignis* **STOP** voran und lassen Sie den Anweisungen ein *Ereignis* **ON** folgen, wie in diesem Beispiel:

```
' Verzweige einmal in der Minute (60 Sekunden) in die
' Unterroutine ZeigZeit:
ON TIMER (60) GOSUB ZeigZeit
' Aktiviere die Verfolgung des 60-Sekunden
' Ereignisses:
```

```
TIMER ON
.
.
.
TIMER STOP    'Verfolgung aussetzen
.
.             ' Eine Abfolge von Zeilen, die nicht
.             ' unterbrochen werden sollen, selbst wenn
.             ' mehr als 60 Sekunden vergangen sind
.
TIMER ON      'Verfolgung wieder aktivieren
.
.
.
END

ZeigZeit:
   ' Lies die aktuelle Zeilen- und Spaltenposition des
   ' Cursors und speichere diese in den Variablen
   ' Zeile und Spalte:
   Zeile = CSRLIN
   Spalte = POS(0)
   ' Gehe in die 24-te Zeile, 20-te Spalte und gib die
   ' Zeit aus:
   LOCATE 24,20
   PRINT TIME$
   ' Bringe den Cursor auf seine vorherige Position
   ' und kehre in das Hauptprogramm zurück:
   LOCATE Zeile,Spalte
RETURN
```

### 6.2.5 Wie Sie Tastenbetätigungen verfolgen

Um eine Tastenbetätigung zu erfassen und die Programmkontrolle in eine Unterroutine für Tastenbetätigung zu leiten, benötigen Sie in Ihrem Programm beide der folgenden Schritte:

**ON KEY**(*Tastennummer*) **GOSUB** *Zeile*
**KEY**(*Tastennummer*) **ON**

Der Wert *Tastennummer* entspricht hier den folgenden Tasten:

| *Wert* | *Taste* |
|---|---|
| 1-10 | Funktionstasten F1-F10 (manchmal als "benutzerdefinierte Tasten" bezeichnet) |
| 11 | NACH OBEN (↑) |
| 12 | NACH LINKS (←) |
| 13 | NACH RECHTS (→) |
| 14 | NACH UNTEN (↓) |
| 15-25 | Benutzerdefinierte Tasten (in den Abschnitten 6.2.5.1 bis 6.2.5.2 finden Sie eine Erläuterung, wie Sie Ihre eigenen Tastenwerte definieren) |
| 30 | Die Funktionstaste F11 (nur bei 101-Tasten-Tastatur) |
| 31 | Die Funktionstaste F12 (nur bei 101-Tasten-Tastatur) |

### Beispiel

Die folgenden beiden Zeilen veranlassen das Programm jedesmal wenn die Funktionstaste F2 betätigt wird, in die Unterroutine `TastSub` zu verzweigen:

```
ON KEY(2) GOSUB TastSub
KEY(2) ON
.
.
.
```

Die folgenden vier Zeilen veranlassen das Programm, in die Unterroutine `UntenTaste` zu verzweigen, wenn die NACH UNTEN-Richtungstaste betätigt wird, und in die Unterroutine `ObenTaste`, wenn die NACH OBEN-Richtungstaste betätigt wird:

```
ON KEY(11) GOSUB ObenTaste
ON KEY(14) GOSUB UntenTaste
KEY(11) ON
KEY(14) ON
.
.
.
```

### 6.2.5.1 Wie Sie eine beliebige benutzerdefinierte Taste verfolgen

Zusätzlich zur Unterstützung der vorbelegten Tastennummern 1-14 (plus 30 und 31 mit der 101-Tasten-Tastatur) ermöglicht BASIC es Ihnen, die Nummern 15-25 jeder der übrigen Tasten auf der Tastatur zuzuweisen. Um Ihre eigenen zu verfolgenden Tasten zu definieren, verwenden Sie diese drei Anweisungen:

**KEY** *Tastennummer*, **CHR$(0) + CHR$(***Abfragecode***)**
**ON KEY(***Tastennummer***) GOSUB** *Zeile*
**KEY(***Tastennummer***) ON**

Hier ist *Tastennummer* ein Wert von 15 bis 25, und *Abfragecode* ist der Abfragecode für die Taste, wie in der Beschreibung der Anweisung **KEY**(*n*) im *Microsoft BASIC-Befehlsverzeichnis* aufgeführt. Zum Beispiel veranlassen die folgenden Zeilen das Programm, jedesmal in die Unterroutine `TTaste` zu verzweigen, wenn der Benutzer die "t"-Taste betätigt:

```
' Definiere Taste 15 (der Abfragecode für "t" ist
' dezimal 20):
KEY 15, CHR$(0) + CHR$(20)
' Definiere die Verfolgung (Zielzeile, wenn "t"
' betätigt wird):
ON KEY(15) GOSUB TTaste
KEY(15) ON                  ' Schalte Erfassung von
                            ' Taste 15 ein.
PRINT "Betätigen Sie b zum Beenden."
DO                        ' Leere Schleife: warte, bis
LOOP UNTIL INKEY$ = "b"   ' Benutzer "b" betätigt.
END
TTaste:             ' Unterroutine zur Tastenbehandlung
   PRINT "t betätigt."
RETURN
```

### 6.2.5.2 Wie Sie benutzerdefinierte umgeschaltete Tasten verfolgen

Sie können auch eine Verfolgung für "umgeschaltete" Tasten durchführen. Eine Taste befindet sich im umgeschalteten Zustand, wenn Sie diese gleichzeitig mit einer oder mehreren der Sondertasten UMSCHALTTASTE, STRG-TASTE oder ALT-TASTE betätigen, oder wenn Sie diese nach dem Einschalten von NUM-FESTSTELLTASTE oder UMSCHALT-FESTSTELLTASTE betätigen.

Nachfolgend wird gezeigt, wie die folgenden Tastenkombinationen verfolgt werden:

**KEY** *Tastennummer*, **CHR$**(*Tastenflagge*) + **CHR$**(*Abfragecode*)
**ON KEY**(*Tastennummer*) **GOSUB** *Zeile*
**KEY**(*Tastennummer*) **ON**

An dieser Stelle ist *Tastennummer* ein Wert von 15 bis 25; *Abfragecode* ist der Abfragecode für die erste Taste, und *Tastenflagge* ist die Summe der einzelnen Codes für die betätigten Sondertasten, wie in der folgenden Tabelle gezeigt:

| *Taste* | *Tastencode* |
|---|---|
| UMSCHALTTASTE | 1, 2 oder 3<br>Tastenverfolgung geht davon aus, daß die linke und rechte Umschalttaste dasselbe sind, so daß Sie die UMSCHALTTASTE mit 1 (links), 2 (rechts) oder 3 (links + rechts) verfolgen können. |
| STRG-TASTE | 4 |
| ALT-TASTE | 8 |
| NUM-FESTSTELLTASTE | 32 |
| UMSCHALT-FESTSTELLTASTE | 64 |
| Jede erweiterte Taste auf der 101-Tasten-Tastatur (mit anderen Worten, eine Taste wie die NACH LINKS-Richtungstaste oder ENTF-TASTE, die sich nicht auf dem Zehnerblock befindet) | 128 |

Zum Beispiel schalten die folgenden Anweisungen die Verfolgung von STRG-TASTE + S ein. Beachten Sie, daß diese Anweisungen so gestaltet sind, daß sowohl die Kombination STRG-TASTE + s (kleines s) als auch STRG-TASTE + S (großes S) verfolgt werden. Um den Großbuchstaben S zu verfolgen, müssen Sie mit Großbuchstaben umgehen, die durch das Halten der UMSCHALTTASTE erstellt werden, genauso wie mit Großbuchstaben, die erstellt werden, wenn die UMSCHALT-FESTSTELLTASTE aktiviert ist, wie hier gezeigt:

```
' 31 = Tastaturabfragecode für die S-Taste
'  4 = Code für die STRG-Taste
KEY 15, CHR$(4) + CHR$(31)      ' Verfolge STRG+s.
' 5 = Code für STRG-Taste + Code für UMSCHALTTASTE
KEY 16, CHR$(5) + CHR$(31)      ' Verfolge STRG+UMSCHALT+s.
```

```
' 68 = Code für STRG-Taste + Code für UMSCHALT-
' FESTSTELLTASTE
KEY 17, CHR$(68) + CHR$(31)     ' Verfolge STRG+UMSCHALT-
                                ' FESTSTELL+s.
ON KEY (15) GOSUB StrgSVerfolg ' Teile Programm mit,
                                ' wohin verzweigt
                                ' werden soll.
ON KEY (16) GOSUB StrgSVerfolg ' (Beachte: dieselbe
                                ' Unterroutine für jede
                                ' Taste)
ON KEY (17) GOSUB StrgSVerfolg
KEY (15) ON          ' Aktiviere Tastenerfassung für
                     ' alle drei Kombinationen.
KEY (16) ON
KEY (17) ON
.
.
.
```

Die folgenden Anweisungen schalten die Verfolgung von STRG+ALT+ENTF ein:

```
' 12 = 4 + 8 = (Code für STRG-Taste) + (Code für
' ALT-Taste)
' 83 = Tastaturabfragecode für ENTF-Taste
KEY 20, CHR$(12) + CHR$(83)
ON KEY(20) GOSUB TastBehand
KEY (20) ON
.
.
.
```

Beachten Sie in dem vorhergehenden Beispiel, daß die BASIC-Ereignisverfolgung die normale Bedeutung von STRG+ALT+ENTF (Warmstart des Systems) überschreibt. Eine solche Verfolgung in Ihrem Programm bietet eine einfache Möglichkeit, den Benutzer vor zufälligem Neustart der Maschine zu schützen, während ein Programm läuft.

Wenn Sie eine 101-Tasten-Tastatur verwenden, können Sie jede Taste auf dem erweiterten Tastaturblock durch Zuweisung der Zeichenkette

**CHR$(128) + CHR$(***Tastaturabfragecode***)**

zu jedem Wert für *Tastennummer* von 15 bis 25, verfolgen.

Das nächste Beispiel zeigt, wie Sie die NACH LINKS-Richtungstaste sowohl auf dem erweiterten Cursor-Tastatur-Block als auch auf dem Zehnerblock verfolgen können.

```
' 128 = Tastaturflagge für Tasten auf dem erweiterten
' Cursor-Tastaturblock
' 75 = Tastaturabfragecode für die  Pfeiltaste
KEY 15, CHR$(128) + CHR$(75) ' Verfolge Taste auf
                             ' dem erweiterten Cursor-
                             ' Tastaturblock
ON KEY(15) GOSUB CursorBlock
KEY(15) ON
ON KEY(12) GOSUB ZehnerBlock ' Verfolge Taste auf
                             ' dem Zehnerblock
KEY(12) ON
DO: LOOP UNTIL INKEY$ = "b"  ' Leere Schleife
END
CursorBlock:
   PRINT "   -Taste wurde auf dem Cursorblock betätigt."
RETURN
ZehnerBlock:
   PRINT "  -Taste wurde auf dem Zehnerblock betätigt."
RETURN
```

## 6.2.6 Wie Sie Musikereignisse verfolgen

Wenn Sie die Anweisung **PLAY** zum Spielen von Musik verwenden, können Sie wählen, ob die Musik im Vordergrund oder im Hintergrund spielt. Wenn Sie Vordergrundmusik wählen, was der Standardeinstellung entspricht, kann nichts anderes ausgeführt werden, bis die Musik beendet ist. Wenn Sie jedoch die Option **MB** (Music Background) in einer **PLAY** Musikzeichenkette verwenden, spielt die Melodie im Hintergrund, während nachfolgende Anweisungen Ihres Programmes ausgeführt werden.

Die **PLAY**-Anweisung spielt Musik im Hintergrund, indem sie den Puffer mit bis zu 32 Noten auffüllt, dann die Noten in dem Puffer spielt, während das Programm andere Dinge ausführt. Eine "Musik-Verfolgung" funktioniert, indem sie die Anzahl der aktuell im Puffer zum Spielen übrig gebliebenen Noten prüft. Sobald diese Anzahl unter die Grenze fällt, die Sie in der Verfolgung gesetzt haben, verzweigt das Programm zu der ersten Zeile der angegebenen Unterroutine.

Um in Ihrem Program eine Musik-Verfolgung einzurichten, benötigen Sie die folgenden Anweisungen:

**ON PLAY**(*Grenze*) **GOSUB** *Zeile*
**PLAY ON**
**PLAY "MB"**
.
.
.
**PLAY** *Musikzeichenkette*
[**PLAY** *Musikzeichenkette*]
.
.
.

An dieser Stelle ist *Grenze* eine Zahl zwischen 1 und 32. Zum Beispiel veranlaßt dieser Ausschnitt das Programm immer dann in die Unterroutine `MusikVerfolg` zu verzweigen, wenn die Anzahl der im Musikpuffer verbliebenen Noten von acht auf sieben sinkt:

```
ON PLAY (8) GOSUB MusikVerfolg
PLAY ON
.
.
.
PLAY "MB"    ' Spiele nachfolgende Noten im Hintergrund.
PLAY "o1 A# B# C-"
.
.
.
MusikVerfolg:
      'Unterroutine zur Musikverfolgung
RETURN
```

***Wichtig*** Der Auslöser für eine Musikverfolgung ist die von *Grenze* auf *Grenze* -1 sinkende Anzahl an Noten. Wenn zum Beispiel der Musikpuffer im vorhergehenden Beispiel niemals mehr als sieben Noten enthält, würde die Verfolgung niemals stattfinden. In dem Beispiel wird die Verfolgung nur vorgenommen, wenn die Anzahl an Noten von acht auf sieben sinkt.

Sie können eine Unterroutine zur Musikverfolgung verwenden, um dasselbe Musikstück während der Programmausführung wiederholt zu spielen, wie in dem nächsten Beispiel gezeigt:

```
' Schalte die Verfolgung der Hintergrundmusik-
' Ereignisse ein:
PLAY ON
' Verzweige zu der Unterroutine Auffrischen, wenn sich
' weniger als zwei Noten im Puffer der
' Hintergrundmusik befinden:
ON PLAY(2) GOSUB Auffrischen
PRINT "Mit beliebiger Taste starten, b zum Beenden."
Pause$ = INPUT$(1)
' Wähle für PLAY die Option Hintergrundmusik:
PLAY "MB"
' Beginne, die Musik zu spielen, so daß Noten in den
' Puffer der Hintergrundmusik geschrieben werden:
GOSUB Auffrischen
I = 0
DO
   ' Gibt die Zahlen von 0 bis 10.000 solange aus, bis
   ' der Benutzer die "b"-Taste betätigt. Während dies
   ' passiert, wird die Musik im Hintergrund
   ' wiederholt:
   PRINT I
   I = (I + 1) MOD 10001
LOOP UNTIL INKEY$ = "b"
END
Auffrischen:
   ' Spielt den Eröffnungssatz von Beethovens Fünfter
   ' Sinfonie:
   Beet$    = "t180 o2 p2 p8 L8 GGG L2 E-"
   Hoven$   = "p24 p8 L8 FFF L2 D"
   PLAY Beet$ + Hoven$
RETURN
```

## 6.3 Fehler- und Ereignisverfolgung in SUB- oder FUNCTION-Prozeduren

Der wichtigste im Gedächtnis zu behaltende Aspekt bei der Verwendung von Fehler- oder Ereignisverfolgung mit BASIC-Prozeduren ist, daß es beiden der folgenden Anweisungen erlaubt ist, innerhalb eines **SUB...END SUB**- oder **FUNCTION...END FUNCTION**-Blockes zu erscheinen:

**ON ERROR GOTO** *Zeile*
**ON** *Ereignis* **GOSUB** *Zeile*

Die *Zeile*, auf die in jedem Fall Bezug genommen wird, muß jedoch eine Zeile im Modul-Ebenen-Code kennzeichnen, nicht eine andere Zeile innerhalb der **SUB** oder **FUNCTION**.

### Beispiel

Das folgende Beispiel zeigt, wo eine Fehlerbehandlungsroutine plaziert wird, die Fehler verarbeitet, die innerhalb einer **SUB**-Prozedur verfolgt werden.

```
CALL KurzSub
END

AbfangFehler:
   ' Plaziere die Routine AbfangFehler auf der Modul-
   ' Ebene außerhalb des Unterprogrammes:
   PRINT "Fehler" ERR ", abgefangen mit Fehlerbehandlung."
RESUME NEXT

SUB KurzSub STATIC
   ON ERROR GOTO AbfangFehler
   ERROR 62
END SUB
```

### *Ausgabe*

```
Fehler 62, abgefangen mit Fehlerbehandlung.
```

## 6.4 Verfolgung in mehrmoduligen Programmen

Ein wichtiger Unterschied zwischen Ereignisverfolgung und Fehlerverfolgung wird deutlich, wenn Ihr Programm aus mehr als einem Modul besteht. Ereignisse können quer durch Module verfolgt werden; das heißt, nachdem Sie in einem Modul eine Ereignisverfolgung aktiviert sowie eine Unterroutine zur Ereignisbehandlung definiert haben, löst das Auftreten dieses Ereignisses in anderen Modulen eine Verzweigung in diese Unterroutine aus.

Jedes Modul, das Fehler verfolgt, muß jedoch zur Aktivierung der Fehlerverfolgung seine eigene **ON ERROR GOTO**-Anweisung haben. Darüberhinaus muß jedes Modul über seine eigene Fehlerbehandlungsroutine verfügen.

### 6.4.1 Ereignisverfolgung quer durch Module

Die Ausgabe des folgenden Programmes zeigt, daß eine in dem Hauptmodul für die Funktionstaste F1 gesetzte Verfolgung sogar dann ausgelöst wird, wenn sich die Programmkontrolle in einem anderen Modul befindet:

```
' ====================================================
'                          HAUPTMODUL
' ====================================================
ON KEY (1) GOSUB F1Taste
KEY (1) ON
PRINT "Im Hauptmodul. Betätigen Sie f zum Fortfahren."
DO : LOOP UNTIL INKEY$ = "f"
CALL SubTaste
PRINT "Zurück im Hauptmodul. Betätigen Sie b zum Beenden."
DO : LOOP UNTIL INKEY$ = "b"
END
F1Taste:
  PRINT "Bearbeitung der F1-Tastenbetätigung im Hauptmodul."
RETURN
' ====================================================
'                          MODUL SUBTASTE
' ====================================================
SUB SubTaste STATIC
   PRINT "Im Modul mit SUBTASTE. Betätigen Sie z, ";
   PRINT "um zurückzukehren."
   ' Die Betätigung von F1 an dieser Stelle ruft immer
   ' noch die Unterroutine F1Taste im Hauptmodul auf:
   DO : LOOP UNTIL INKEY$ = "z"
END SUB
```

***Ausgabe***

```
Im Hauptmodul. Betätigen Sie f zum Fortfahren.
Bearbeitung der F1-Tastenbetätigung im Hauptmodul.
Im Modul mit SUBTASTE. Betätigen Sie z, um zurückzukehren.
Bearbeitung der F1-Tastenbetätigung im Hauptmodul.
Zurück im Hauptmodul. Betätigen Sie b zum Beenden.
Bearbeitung der F1-Tastenbetätigung im Hauptmodul.
```

## 6.4.2 Fehlerverfolgung in mehrmoduligen Programmen

Anders als Ereignisverfolgung ist Fehlerverfolgung lokal zu jedem Modul. Das heißt, eine Fehlerbehandlungsroutine in einem Modul kann nicht durch das Auftreten eines Fehlers in einem anderen Modul aufgerufen werden. Die folgenden Regeln beschreiben, wie Fehler in einem aus mehreren Modulen aufgebauten Programm verfolgt werden:

1. Wenn in einem Modul Fehler verfolgt werden sollen, dann muß dieses Modul seine eigene Fehlerbehandlungsroutine haben.
2. Jede Routine kann nur solche Fehler behandeln, die in dem Modul auftreten, in dem diese vorkommt.
3. Fehlerverfolgung muß in jedem Modul mit einer separaten **ON ERROR GOTO**-Anweisung eingeschaltet werden.

   Beachten Sie, daß **ON ERROR GOTO** ausgeführt sein muß, bevor es Auswirkungen hat. Sie müssen die Anweisung **ON ERROR GOTO** daher dort anordnen, wo der Ablauf einer Programmkontrolle diese erreichen kann; plazieren Sie sie zum Beispiel innerhalb einer Prozedur, die vom Hauptmodul aufgerufen wird (wie Sie es dem folgenden Beispiel entnehmen können).

Diese Modularität der Fehlerverfolgung ist wirklich ganz leicht zu handhaben, besonders wenn Sie eine Fehlerbehandlungsroutine in einer Quick-Bibliothek ablegen möchten. Jede Bibliothek kann ihre eigene entsprechende Fehlerverfolgung besitzen.

### Beispiel

Das folgende Beispiel zeigt, wie Fehler für eine Prozedur in einer Quick-Bibliothek verfolgt werden. Die Prozedur `AdapterTyp` in dieser Bibliothek prüft mit allen **SCREEN**-Anweisungen, die möglich sind, welche grafischen Bildschirmmodi Ihre Computerhardware unterstützt. (Weitere Informationen zu Quick-Bibliotheken finden Sie in Kapitel 8, "Quick-Bibliotheken", in *Lernen und Anwenden von Microsoft QuickBASIC*.)

```
' ======= MODUL-EBENEN-CODE IN QUICK-BIBLIOTHEK ======
' Die Fehlerbehandlungsroutinen für Prozeduren in
' dieser Bibliothek müssen auf dieser Ebene definiert
' sein. Diese Ebene wird nur ausgeführt, wenn in den
' Prozeduren ein Fehler auftritt.
' ======================================================
'
DEFINT A-Z
CONST FALSCH = 0, WAHR = NOT FALSCH
RetteMich:        ' Fehlerbehandlungsroutine
   AnzeigeFehler = WAHR
   RESUME NEXT
'
' ===== PROZEDUR-EBENEN-CODE IN QUICK-BIBLIOTHEK =====
'   Fehlerverfolgung wird auf dieser Ebene aktiviert.
' ======================================================
'
SUB AdapterTyp STATIC
   SHARED AnzeigeFehler    ' Die Variable AnzeigeFehler
                           ' wird gemeinsam mit der
                           ' Fehlerroutine RetteMich im
                           ' obigen Modul-Ebenen-Code
                           ' benutzt.
   DIM MonitorTyp(1 TO 13)' Dimensioniere Datenfeld
                           ' MonitorTyp.
   J = 1                   ' Initialisiere Indexzähler
                           ' für Datenfeld MonitorTyp.
   ON ERROR GOTO RetteMich ' Schalte Fehlerverfolgung
                            ' ein.
   FOR Test = 13 TO 1 STEP -1
      SCREEN Test      ' Versuche eine SCREEN-Anweisung
                       ' um festzustellen, ob sie
                       ' einen Fehler verursacht.
      IF NOT AnzeigeFehler THEN  ' Kein Fehler, dies
                                 ' ist also ein
                                 ' gültiger
                                 ' Bildschirmmodus.
         MonitorTyp(J) = Test    ' Speichere den Modus
                                 ' in dem Datenfeld.
         J = J + 1       ' Zähle den Indexzähler hoch.
      ELSE
         AnzeigeFehler = FALSCH  ' Fehler, also setze
                                 ' AnzeigeFehler
      END IF                     ' zurück.
   NEXT Test
```

```
    SCREEN 0, 0        ' Setze 80-Spalten Textmodus.
    WIDTH 80
    LOCATE 5, 10
    PRINT "Ihr Computer unterstützt die folgenden";
    PRINT " Bildschirmmodi:"
    PRINT
    FOR I = 1 TO J      ' Gib die Modi aus, die keine
                        ' Fehler verursachen.
       PRINT TAB(20); "BILDSCHIRM"; MonitorTyp(I)
    NEXT I
    LOCATE 20, 10
    PRINT "Weiter mit beliebiger Taste..."
    DO: LOOP WHILE INKEY$ = ""
END SUB
```

Wenn Sie möchten, daß die Fehlerbehandlungsroutine in jedem Modul exakt dasselbe tut, können Sie jede Routine dazu veranlassen, eine gemeinsame fehlerverarbeitende SUB-Prozedur aufzurufen, wie in dem nächsten Beispiel gezeigt:

```
' ====================================================
'                      HAUPTMODUL
' ====================================================
'
DECLARE SUB GlobalBehandlung (ModulName$)
DECLARE SUB KurzSub ()
ON ERROR GOTO LokalBehandlung
ERROR 57              ' Simuliere Auftreten des Fehlers
                      ' 57 ("Geräte-E/A-Fehler").
KurzSub               ' Rufe die SUB KurzSub auf.
END
LokalBehandlung:
   ModulName$ = "HAUPT"
   GlobalBehandlung ModulName$ ' Rufe die SUB
                               ' GlobalBehandlung auf.
RESUME NEXT
'
' ====================================================
'                  MODUL KURZSUB
' ====================================================
'
DECLARE SUB GlobalBehandlung (ModulName$)
```

```
LokalBehandlung:
   ModulName$ = "KURZSUB"
   GlobalBehandlung ModulName$ ' Rufe die SUB
                               ' GlobalBehandlung auf.
RESUME NEXT
SUB KurzSub STATIC
   ON ERROR GOTO LokalBehandlung
   ERROR 13            ' Simuliere den Fehler
                       ' "Unverträgliche Datentypen".
END SUB
'
' ======================================================
'               MODUL GLOBALBEHANDLUNG
' ======================================================
'
SUB GlobalBehandlung (ModulName$) STATIC
   PRINT "Verfolgter Fehler"; ERR; "im Modul ";
   PRINT ModulName$; "."
END SUB
```

***Ausgabe***

```
Verfolgter Fehler 57 im Modul HAUPT.
Verfolgter Fehler 13 im Modul KURZSUB.
```

## 6.5 Fehler- und Ereignisverfolgung in Programmen, die mit bc kompiliert sind

Wenn beide der folgenden Aussagen für Ihr Programm zutreffen, dann müssen Sie zum Kompilieren Ihrer Programme die entsprechende unten aufgeführte **bc**-Befehlszeilenoption verwenden:

- Sie entwickeln Ihr Programm außerhalb der interaktiven QuickBASIC-Umgebung; wenn Sie also Ihren eigenen Texteditor zum Schreiben Ihres BASIC-Quellcodes verwenden und anschließend mit den Befehlen **bc** und **link** ein selbständig lauffähiges Programm aus diesem Quellcode erzeugen.

- Ihr Programm enthält eine der Anweisungen, die unten in der zweiten Spalte aufgeführt sind.

| *Option der Befehlszeile* | *Anweisungen* |
|---|---|
| /e | **ON ERROR GOTO**<br>**RESUME** *Zeile*<br>Die Option /e zeigt dem Compiler an, daß Ihr Programm seine eigene Fehlerverfolgung mit **ON ERROR GOTO**- und **RESUME**-Anweisungen durchführt. |
| /v | **ON** *Ereignis* **GOSUB**<br>*Ereignis* **ON**<br>Die Option /v zeigt dem Programm an, zwischen jeder Anweisung zu prüfen, ob das angegebene *Ereignis* aufgetreten ist. (Vergleichen Sie dies mit der Auswirkung von /w.) |
| /w | **ON** *Ereignis* **GOSUB**<br>*Ereignis* **ON**<br>Die Option /w zeigt dem Programm an, zwischen jeder Zeile zu prüfen, ob das angegebene *Ereignis* aufgetreten ist. (Vergleichen Sie dies mit der Auswirkung von /v.)<br>Da BASIC mehrere Anweisungen auf einer Zeile zuläßt, prüfen mit /w kompilierte Programme weniger häufig als solche, die mit /v kompiliert sind. |
| /x | **RESUME**<br>**RESUME NEXT**<br>**RESUME 0**<br>Die Option /x zeigt dem Compiler an, daß Sie in Ihrem Programm eine der obigen Formen von **RESUME** verwendet haben, um die Kontrolle aus einer Fehlerbehandlungsroutine zurückzugeben. |

## Beispiel

Die folgenden DOS-Befehlszeilen kompilieren und binden ein als `RMTAB.BAS` benanntes Modul und erzeugen ein als `RMTAB.EXE` benanntes selbständiges Programm. (Da dieses Programm ohne die Option /o kompiliert ist, erfordert es noch die Laufzeit-Bibliothek **BRUN40.LIB**, um lauffähig zu sein. Weitere Informationen zum Kompilieren und Binden finden Sie in Kapitel 9, "Wie Sie aus DOS heraus kompilieren und binden", in *Lernen und Anwenden von Microsoft QuickBASIC*.)

Da dieses Modul **ON ERROR GOTO**- und **RESUME NEXT**-Anweisungen enthält, sind für das Kompilieren die Optionen **/e** und **/x** erforderlich.

```
bc rmtab , , /e /x;
link rmtab;
```

## 6.6 Beispielanwendung: Verfolgen von Dateizugriffsfehlern (DATFEHL.BAS)

Das folgende Programm erhält als Eingabe sowohl einen Dateinamen als auch eine Zeichenkette, die in der Datei gesucht werden soll. Es listet dann alle Zeilen der gegebenen Datei auf, die die angegebene Zeichenkette enthalten. Wenn ein Dateizugriffsfehler auftritt, wird dieser mit der Routine `FehlerProz` verfolgt und bearbeitet.

### Verwendete Anweisungen und Funktionen

Dieses Programm demonstriert die Verwendung der folgenden Anweisungen und Funktionen zur Fehlerbehandlung:

- **ERR**
- **ON ERROR GOTO**
- **RESUME**
- **RESUME NEXT**

### Programm-Listing

```
' Deklariere die im Programm verwendeten symbolischen
' Konstanten:
CONST FALSCH = 0, WAHR = NOT FALSCH
DECLARE FUNCTION LiesDateiName$ ()
' Schalte die FEHLER-Verfolgung ein und gib den Namen
' der Fehlerbehandlungsroutine an:
ON ERROR GOTO FehlerProz
```

```
DO
   Neustart = FALSCH
   CLS
   DateiName$ = LiesDateiName$    ' Eingabe des
                                  ' Dateinamens.
   IF DateiName$ = "" THEN
      END                         ' Beende, wenn die
                                  ' <EINGABETASTE>
                                  ' betätigt wird.
   ELSE
      ' Anderenfalls öffne die Datei; weise ihr dabei
      ' die nächste verfügbare Dateinummer zu
      DateiNum = FREEFILE
      OPEN DateiName$ FOR INPUT AS DateiNum
   END IF
   IF NOT Neustart THEN
      ' Gib zu suchende Zeichenkette ein:
      PRINT "Geben Sie die zu suchende Zeichenkette ";
      LINE INPUT "ein: ", SuchKette$
      SuchKette$ = UCASE$(SuchKette$)
      ' Durchlaufe die Zeilen der Datei und gib diese
      ' aus, wenn sie die zu suchende Zeichenkette
      ' enthalten:
      ZeileNum = 1
      DO WHILE NOT EOF(DateiNum)
         ' Einlesen einer Zeile aus der Datei:
         LINE INPUT #DateiNum, ZeilePuffer$
         ' Prüfe auf Zeichenkette und gib die Zeile
         ' sowie deren Nummer aus, wenn gefunden:
         IF INSTR(UCASE$(ZeilePuffer$), SuchKette$) <> 0_
           THEN
            PRINT USING "#### &"; ZeileNum, ZeilePuffer$
         END IF
         ZeileNum = ZeileNum + 1
      LOOP
      CLOSE DateiNum       ' Schließe die Datei.
   END IF
LOOP WHILE Neustart = WAHR
END
```

```
FehlerProz:
   SELECT CASE ERR
      CASE 64:                ' Unzulässiger Dateiname
         PRINT "** FEHLER - Unzulässiger Dateiname"
         ' Lies einen neuen Dateinamen und versuche
         ' erneut:
         DateiName$ = LiesDateiName$
         ' Fahre mit der Anweisung fort, die den
         ' Fehler verursachte:
         RESUME
      CASE 71:                ' Diskette nicht bereit
         PRINT "** FEHLER - Diskette nicht bereit"
         PRINT "Betätige F zum Fortfahren, N zum ";
         PRINT "Neustart, B zum Beenden: "
         DO
           Zeich$ = UCASE$(INPUT$(1))
           IF Zeich$ = "F" THEN
              RESUME          ' Fahre dort fort, wo
                              ' beendet wurde.
           ELSEIF Zeich$ = "N" THEN
              Neustart = WAHR   ' Fahre am Beginn
              RESUME NEXT       ' fort.
           ELSEIF Zeich$ = "B" THEN
              END          ' Fahre überhaupt nicht fort.
           END IF
         LOOP
      CASE 53, 76:               ' Datei oder Pfad
                                 ' nicht gefunden.
         PRINT "** FEHLER - Datei oder Pfad nicht gefunden"
         DateiName$ = LiesDateiName$
         RESUME
      CASE ELSE:             ' Unvorhergesehener Fehler
         ' Schalte Fehlerverfolgung aus und gib
         ' Standard-Systemmeldung aus:
         ON ERROR GOTO 0
   END SELECT
```

```
'
' ================== LIESDATEINAME$ ==================
'   Gibt eine vom Benutzer eingegebene Datei zurück
' ====================================================
'
FUNCTION LiesDateiName$ STATIC
   PRINT "Gib zu durchsuchende Datei ein ";
   PRINT "(betätige EINGABETASTE zum Beenden): "
   INPUT DTemp$
   LiesDateiName$ = DTemp$
END FUNCTION
```

## 6.7 Zusammenfassung der Anweisungen und Funktionen zur Verfolgung

Die folgende Tabelle listet die Anweisungen und Funktionen auf, die von BASIC zur Verfolgung und Bearbeitung von Fehlern und Ereignissen verwendet werden.

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Verfolgen von Fehlern, während ein Programm läuft | **ON ERROR GOTO** *Zeile* | Veranlaßt ein Programm, in die angegebene *Zeile* zu verzweigen, wobei *Zeile* sich auf eine Zeilennummer oder Zeilenmarke bezieht. Die Verzweigung findet während der Ausführung des Programmes jedesmal statt, wenn ein Fehler auftritt. |
| | **RESUME** | Gibt die Kontrolle nach Ausführung einer Fehlerbehandlungsroutine an das Programm zurück. Das Programm fährt fort entweder mit der Anweisung, die den Fehler verursachte (**RESUME** [0]), mit der Anweisung hinter derjenigen, die den Fehler verursachte (**RESUME NEXT**), oder mit der von *Zeile* angegebenen Zeile (**RESUME** *Zeile*). |

| *Aufgabe* | *Anweisung oder Funktion* | *Funktion* |
|---|---|---|
| Lesen von Informationen zum Fehlerstatus | **ERR** | Gibt den Code für einen Fehler zurück, der während der Laufzeit auftritt |
| | **ERL** | Gibt die Nummer der Zeile an, in der der Fehler auftrat (hilfreich nur dann, wenn das Programm Zeilennummern hat) |
| | **ERDEV** | Gibt einen gerätespezifischen Fehlercode für das letzte Gerät (wie einen Drucker) an, auf dem DOS einen Fehler entdeckt hat |
| | **ERDEV$** | Gibt den Namen des letzten Gerätes zurück, auf dem DOS einen Fehler entdeckt hat |
| Definieren Ihres eigenen Fehlercodes | **ERROR** | Simuliert das Auftreten eines BASIC-Fehlers; kann ebenso dazu verwendet werden, einen Fehler zu definieren, der nicht von BASIC verfolgt wird |
| Verfolgen von Ereignissen, während der Programmausführung | **ON** *Ereignis* **GOSUB** *Zeile* | Verursacht eine Verzweigung in die mit *Zeile* beginnende Unterroutine, wobei *Zeile* sich entweder auf eine Zeilennummer oder Zeilenmarke bezieht. Die Verzweigung wird immer dann ausgeführt, wenn das angegebene *Ereignis* während der Ausführung auftritt. |
| | *Ereignis* **ON** | Schaltet Verfolgung des angegebenen *Ereignisses* ein |
| | *Ereignis* **OFF** | Schaltet Verfolgung des angegebenen *Ereignisses* aus |
| | *Ereignis* **STOP** | Setzt Verfolgung des angegebenen *Ereignisses* aus |
| | **RETURN** | Gibt die Kontrolle nach Ausführung einer Ereignisbehandlungsroutine an das Programm zurück. Das Programm fährt entweder mit der Anweisung fort, die direkt der Stelle in dem Programm folgt, an der das Ereignis auftrat (**RETURN**), oder mit der Zeile, die von *Zeile* angegeben wird (**RETURN** *Zeile*). |

# Index


' (Apostroph), Einleiten von Kommentaren xvi
$ (Dollarzeichen), Suffix für Zeichenkettentyp 4.3
* (Stern), Zeichenkette fester Länge mit AS 4.3, 4.5
+ (Pluszeichen), Zusammensetzen von Zeichenketten 4.6

A

Ablaufsteuerung, Strukturen zur
  Definition 1.2
  Einrückung xvi
  Entscheidung 1.6
  neue 1.2
  Schleife 1.19
  verwendet in BASIC 1.40
Absolute Koordinaten *Siehe* Koordinaten, absolute
Abtastcode, zur Tastenverfolgung 6.15
Adreßgleiche Variablen 2.39
Alphabetisierung von Zeichenketten 4.8
AND Operator 1.4
AND Option mit Grafikanweisung PUT 5.62
Animation
  Bildflimmern reduzieren 5.69
  Bildschirmseiten 5.71
  einfache Grafikanweisungen 5.58
  GET und PUT 5.58, 5.65
  PALETTE USING 5.40
Anweisungen *Siehe* entsprechende Anweisung
Apostroph (')
  eingeben xiii
  Kommentare einleiten xvi
Argumente
  Übereinstimmung mit Parametern 2.23
Argumente (*Fortsetzung*)
  übergeben
    als Referenz 2.7, 2.29
    als Wert 2.7, 2.29, 2.30
  übergeben an SUB-Prozeduren
    CALL, mit 2.13
    CALL, ohne 2.13
  überprüfen auf korrekten Typ und korrekte Anzahl 2.23
  unterschieden von Parametern 2.14
ASC-Funktion 4.4
ASCII
  Dateien, gelesen als sequentielle Dateien 3.24
  Zeichencodes
    Alphabetisierung von Zeichenketten 4.8
    Argumente für die Funktion CHR$ 4.4
    festlegen mit der Funktion ASC 4.4
    Zeichen im Hauptspeicher speichern 4.4
Attribute
  Bildschirmmodus 1 5.35
  EGA-Bildschirmmodi 5.36, 5.39
  VGA-Bildschirmmodi 5.39
  zuweisen verschiedener Farben 5.39
Ausdrücke
  Boolesche 1.3
  falsche 1.4
  Übergabe an Prozeduren 2.17
  wahre 1.4
  Zeichenkette 4.3
  Zeichenkette *Siehe auch* Zeichenketten
Ausfüllen *Siehe* Ausmalen
Ausfüllen mit Mustern *Siehe auch* Ausmalen
Ausgabebereich 3.5, 3.29
Ausmalen
  Abbildungen
    außerhalb 5.42
    innerhalb 5.42
  andere Farbe als für den Rand 5.43
  Definition einer monochromen Musterkachel 5.45
  dieselbe Farbe wie der Rand 5.44
  Farben 5.42
  farbige Musterkachel, Definition 5.48, 5.52
  Muster 5.44, 5.73, 5.86
Automatische Variablen 2.10, 2.40

B

BASIC, Bücher über xi
Baud-Rate 3.50
Benannter COMMON-Block 2.38, 2.46
Bewegen von Bildern mit PUT 5.61
Bilder
  kopieren auf den Bildschirm mit PUT 5.58
  speichern in den Hauptspeicher mit GET 5.60
Bildpunkte
  Definition 5.5
  positionieren mit Koordinaten 5.5
  PRESET, zeichnen mit 5.7
  PSET, zeichnen mit 5.7
  Textcursor 3.16
Bildschirmauflösung, Anpassung mit SCREEN 5.4
Bildschirmkonfiguration
  Grafikmodus 5.4
  Textmodus 3.4
Bildschirmmodi
  auswählen 5.4
  prüfen auf 5.4

Bildschirmseiten
  aktive Seite
    auswählen 5.71
    Standardwert 5.72
  sichtbare Seite
    auswählen 5.71
    Standardwert 5.72
Binär
  Eingabe 3.32
  Suche, Beispiel 3.60
  Zahlen, umwandeln in hexadezimal 5.46
  Zugriff, verglichen mit Direktzugriff 3.43
Binärzugriffsdatei
  erstellen 3.21, 3.43
  lesen 3.44
  öffnen 3.21, 3.43
  schreiben in 3.44
Bit-Ebenen 5.52
Bitweise Operatoren, NOT 1.5
  *Siehe auch* Logische Operatoren
Bogen
  definieren 5.17
  zeichnen 5.17, 5.20
Bogenmaß
  Definition 5.17
  im Unterschied zu Grad 5.17
Boolesche
  Ausdrücke
    Definition 1.3
    logische Operatoren 1.4
    vergleichen von anderen Ausdrücken 1.4
    Vergleichsoperatoren 1.3
  Konstanten 1.6

C

CALL-Anweisung, Aufruf einer SUB-Prozedur 2.13, 2.24
CASE-Klausel *Siehe* SELECT CASE-Anweisung
CGA
  Bildschirmmodi, unterstützt von 5.35
  wechseln der Palette 5.36
CHAIN-Anweisung 2.45
CHR$-Funktion 4.4
CIRCLE-Anweisung
  Bögen 5.17, 5.20
  Ellipsen 5.15
  Kreise 5.14
  Tortendiagramme 5.21
CLEAR-Anweisung 2.44
COLOR-Anweisung
  Hintergrundfarbe, Steuerung von 5.36
  Schreibweise in Bildschirmmodus 1 5.36
  verändern der Palette mit 5.36
  Vordergrundfarbe, steuern 5.36
COM1 3.47
COM2 3.47
COMMON-Anweisung
  benannter COMMON-Block 2.38, 2.46
  SHARED-Attribut
    Beschreibung 2.53
    definieren 2.32
    SUB, FUNCTION, teilen mit Variablen 2.53
    Variablen gemeinsam benutzen 2.38, 2.45
    Variablen global machen 2.34
    Variablen zwischen Modulen gemeinsam benutzen 2.38
  unbenannter COMMON-Block 2.46
COMMON-Block
  benannt 2.38, 2.46
  unbenannt 2.46
CONS 3.47
CSRLIN-Funktion 3.18
Cursor
  grafischer
    Definition 5.9
    finden der Position von 5.33
  Text
    Definition 3.15
    finden der Position von 3.18
    positionieren mit LOCATE 3.16
    verändern der Form von 3.17
CV*Typ*-Anweisung 3.35
CV*Typ*MBF-Anweisung 3.35

D

Darstellungsfeld
  grafisches
    VIEW SCREEN, definieren mit 5.26
    VIEW, definieren mit 5.25
    Vorteile 5.25
  Text
    definiert 3.8
    erstellen mit VIEW PRINT 3.8
    rollen geschriebener Ausgabe in 3.9
Datei
  E/A
    Definition 3.19
    Geräte-E/A im Unterschied zu 3.47
  Namen
    gültige Zeichen 3.22
    kürzen 3.23
    OPEN-Anweisung 3.22
  Nummern
    CLOSE, freigeben mit 3.24
    FREEFILE, erhalten mit 3.22
    gültige Werte 3.22
    OPEN-Anweisung 3.22
  Zeiger
    GET oder PUT, bewegen mit 3.43
    SEEK, bewegen mit 3.44
Dateizugriffsmodi
  APPEND 3.21, 3.28
  BINARY 3.22, 3.43
  Direktzugriff 3.33
  INPUT 3.21, 3.27
  OUTPUT 3.21, 3.26
  RANDOM 3.22
Datendateien
  Dateinummern 3.22
  Definition 3.19
  Direktzugriff 3.20
  erstellen 3.20
  hinzufügen von Sätzen
    PRINT# 3.28
    WRITE# 3.28
  lesen
    Sätze, INPUT# 3.31
    Sätze, INPUT$ 3.32
    Sätze, LINE INPUT# 3.31
    Zeichen für Zeichen 3.32
    Zeile für Zeile 3.31
  öffnen 3.20
  schließen
    CLOSE-Anweisung 3.24
    Programmende 3.24
    RUN-Anweisung 3.24
  sequentiell 3.20
  verfolgen von Dateizugriffsfehlern 6.28
  Vorteile von 3.19
  wie sie organisiert sind 3.19
Datendateipuffer 3.24
Datenfelder
  Format
    Argumentenliste 2.15
    FUNCTION-Anweisung 2.18
    Parameterliste 2.15, 2.19
    SHARED-Anweisung 2.32

Datenfelder, Format (*Fortsetzung*)
STATIC-Anweisung 2.41
SUB-Anweisung 2.14, 2.19
Funktion für obere Grenze 2.20
Funktion für untere Grenze 2.20
gemeinsam benutzen mit Prozeduren 2.32
Übergabe an Prozeduren 2.15, 2.19
Datenfeldgrenze, Funktionen
LBOUND 2.20
UBOUND 2.20
DECLARE-Anweisungen
Argumente überprüfen mit 2.22
nicht von QuickBASIC erzeugt 2.23
plazieren in Include-Dateien 2.26
wo erforderlich 2.23
DEF FN-Funktion
FUNCTION-Prozedur verglichen mit 2.7
lokale Variablen in 2.7
DIM-Anweisung
SHARED-Attribut
Beispiel, adreßgleiche Variablen 2.39
Definition 2.32
Variablen gemeinsam benutzen 2.53
Variablen global machen 2.34
wann verboten 2.10
Direktzugriff, verglichen mit Binärzugriff 3.43
Direktzugriffsdateien
Anordnung von Sätzen in 3.24
erstellen 3.21
hinzufügen von Sätzen zu 3.37
lesen von Sätzen aus 3.40
öffnen 3.21, 3.35
sequentielle Dateien verglichen mit 3.20, 3.34
wie Zahlen gespeichert werden in 3.34
DO...LOOP-Anweisung
alternatives Verlassen von 1.32
EXIT DO-Klausel 1.32
Position des Schleifentests 1.28
Schreibweise
Prüfung am Anfang 1.27
Prüfung am Ende 1.28
UNTIL Schlüsselwort 1.31
WHILE Schlüsselwort 1.31
WHILE...WEND, verglichen mit 1.27
Dollarzeichen ($), Suffix für Typ Zeichenkette 4.3
DOS, Bücher über xii
DRAW-Anweisung
Drehen von Abbildungen 5.55, 5.72
Makrobefehle in 5.55
Skalieren von Abbildungen 5.55
Drehen von Abbildungen mit DRAW 5.55, 5.72

E

E/A
Dateien, von und auf 3.19
Geräte, von und auf 3.47
Standard 3.3, 3.11
/e-Compileroption 6.27
EGA, verändern der Palette 5.36, 5.39
Eingabe nach Ende: Fehler 3.30
Eingabe/Ausgabe *Siehe* E/A
Ellipsen, zeichnen 5.14
ELSE-Klausel 1.7
ELSEIF-Klausel 1.10
Entscheidungsstrukturen
Block-IF...THEN...ELSE 1.8
Definition 1.6
einzeiliges IF...THEN...ELSE 1.7
SELECT CASE 1.9, 1.11
EOF-Funktion 3.27, 3.48, 3.65
EQV-Operator 1.4
Ereignis OFF 6.12
Ereignis ON 6.11
Ereignis STOP 6.12
Ereignisbehandlungsroutine
schreiben 6.11
zurückgeben der Kontrolle aus 6.11
Ereigniserfassung
Registrierung 6.9
Verfolgung 6.9
Ereignisregistrierung im Unterschied zu Ereignisverfolgung 6.9
Ereignisverfolgung
*Siehe auch* Ereignis
aktivieren 6.10
ausschalten 6.12
aussetzen 6.12
Befehlszeilenoptionen, die der Compiler erfordert 6.28
Hintergrundmusik 6.18
mehrmodulige Programme 6.22
ON COM 6.11
ON Ereignis GOSUB-Anweisung 6.10
ON KEY 6.11
ON PEN 6.11
ON PLAY 6.11
ON STRIG 6.11
ON TIMER 6.11
Ereignisverfolgung (*Fortsetzung*)
Registrierung im Unterschied zu 6.9
Schreibweise im Unterschied zur Schreibweise von Fehlerverfolgung 6.11
SUB- oder FUNCTION-Prozeduren 6.21
Tastenbetätigungen 6.13
Unterroutine zur Ereignisbehandlung
festlegen 6.10
innerhalb 6.12
schreiben 6.11
zurückkehren aus 6.10
verfolgbare Ereignisse 6.11
Zusammenfassung von Anweisungen und Funktionen 6.31
ERR-Funktion 6.4, 6.28
Ersatzargumente, POS-Funktion 3.18
Erweiterter Grafikadapter *Siehe* EGA
EXIT DEF-Anweisung 2.54
EXIT FUNCTION-Anweisung 2.52
EXIT SUB-Anweisung 2.52

F

Fakultät 2.43
Falsche Ausdrücke 1.4, 3.27
Farbattribute *Siehe* Attribute
Farben
festlegen in Grafikanweisungen 5.35
Hintergrund 5.7
PALETTE USING, verändern mit 5.39
PALETTE, verändern mit 5.39
Relation mit Schärfe 5.35
Verwendung mit CGA 5.34
Vordergrund 5.7
Farbgrafikadapter *Siehe* CGA
Fehlerbehandlungsroutine
Fehler mit ERR identifizieren 6.4
innerhalb der Routine 6.4
spezifizieren 6.4
zurückkehren von 6.6
Fehlerverfolgung
aktivieren 6.3
ausschalten 6.3
Befehlszeilenoptionen, die der Compiler erfordert 6.26
Dateizugriffsfehler 6.28
Fehler mit ERR identifizieren 6.4
Fehlerbehandlungsroutine
innerhalb der Routine 6.4
schreiben 6.4
spezifizieren 6.3
zurückkehren von 6.6

Fehlerverfolgung (*Fortsetzung*)
mehrmodulige Programme 6.23, 6.25
Quick-Bibliotheken 6.23
Schreibweise im Unterschied zur Schreibweise von Ereignisverfolgung 6.11
SUB- oder FUNCTION-Prozeduren 6.21
ungeeignete Bildschirmmodi 5.4
verfolgte Fehler behandeln 6.2
Zusammenfassung von Anweisungen und Funktionen 6.31
Fehlerverfolgung ausschalten 6.3
Felder
Begrenzung in sequentiellen Dateien 3.25, 3.34
Definition 3.19
Sätze für Direktzugriff 3.33
sequentielle Sätze 3.24
Fettgeschriebener Text
markiert Befehlsnamen xiii
markiert Schlüsselwörter xiii
FIELD-Anweisung
definieren von Sätzen für Direktzugriff 3.36
TYPE...END TYPE, im Unterschied zu 3.35, 3.36
FOR...NEXT-Anweisung
EXIT FOR-Klausel 1.24
Programmausführung anhalten 1.25
Schreibweise 1.19
STEP-Schlüsselwort 1.19
verschachteln 1.22
vorzeitiges Verlassen von 1.24
Formatieren von Textausgabe 3.6
Fractal 5.79
FRE-Anweisung 2.45
FREEFILE-Funktion 3.22
FUNCTION-Anweisungen nicht erlaubt in Include-Dateien 2.24
FUNCTION-Prozeduren
aufrufen 2.7, 2.11
DEF FN-Funktion im Unterschied zu 2.7
Ereignisverfolgung in 6.21
Fehlerverfolgung in 6.21
lokale Variablen in 2.7
FUNCTION...END FUNCTION-Prozedur *Siehe* FUNCTION-Prozeduren
Funktionen *Siehe* entsprechende Funktion
Funktionstasten, Verfolgung 6.13

G

Geraden, zeichnen 5.8
Geradengestaltung 5.12
Geräte
COM1 3.47
COM2 3.47
CONS 3.47
E/A-Modi, gültige 3.48
KYBD 3.47
LPT1 3.47, 3.48
LPT2 3.47
LPT3 3.47
SCRN 3.47, 3.48
Geräte-E/A, vergl. mit Datei-E/A 3.47
GET-Anweisung
Datei-E/A
Binärzugriff 3.43
Direktzugriff 3.41, 3.56
Grafiken
Animation 5.65
Berechnung der Datenfeldgröße in 5.60
Bilder in den Speicher kopieren 5.58
Schreibweise 5.58
Globale Variablen
adreßgleiche Variablen 2.39
DEF FN-Funktion 2.7
FUNCTION-Prozedur 2.7
GOSUB-Routine 2.5
SHARED-Attribut 2.34
GOSUB...RETURN-Unterroutine im Unterschied zu SUB-Prozedur 2.6
Grad
umwandeln in Bogenmaß 5.18
verglichen mit Bogenmaß 5.17
Grafische Anweisungen und Funktionen, Zusammenfassung 5.92
Grafische Bildschirmmodi *Siehe* Bildschirmmodi
Grafischer Cursor *Siehe* Cursor, grafischer
Grafisches Darstellungsfeld *Siehe* Darstellungsfeld, grafisches
Großbuchstaben
Dateinamen 3.23
umwandeln in Kleinbuchstaben 4.14

H

Hexadezimale Zahlen 5.46
Hintergrundfarbe
Bildschirmmodus 1 5.36
Standard 5.7

I

IF...THEN...ELSE-Anweisung
Blockform
ELSE-Klausel 1.9
ELSEIF-Klausel 1.9
Schreibweise 1.9
verschachteln 1.11
einzeilige Form 1.7
SELECT CASE-Anweisung im Unterschied zu 1.11
IMP-Operator 1.4
Include-Dateien
COMMON 2.45
Deklarieren von Prozeduren 2.26
nicht erlaubte Anweisungen 2.29
$INCLUDE-Metabefehl
COMMON 2.45
Prozedurdeklarationen 2.26, 2.29
INKEY$ 3.15
INPUT#-Anweisung
Beispiel 3.27
INPUT$ im Unterschied zu 3.32
LINE INPUT# im Unterschied zu 3.31
INPUT$-Funktion
Beispiel 3.50
Datenübertragung per Modem 3.65
INPUT# im Unterschied zu 3.32
Lesen von Daten
Dateien 3.32
Datenübertragungsgerät 3.50
Standardeingabe 3.14, 3.32
LINE INPUT# im Unterschied zu 3.32
INPUT-Anweisung
Anfrage 3.12
Beispiel 3.51
Definition 3.11
Fehlermeldung ungültige Eingabe 3.12
Format der Variablenliste 3.11
LINE INPUT im Unterschied zu 3.12
Unterdrücken von Wagenrücklauf-Zeilenvorschub-Folge nach Eingabe 3.13
INSTR-Funktion 4.8, 4.17

K

Kartesische Koordinaten *Siehe* Logische Koordinaten
Kleinbuchstaben
Dateinamen 3.23
übertragen in Großbuchstaben 4.14

Kommentare, eingeleitet mit Apostroph xiii
Komplexe Zahlen 5.79
Konstanten
  symbolische xv
  Übergabe an Prozeduren 2.16
  Zeichenkette, literale 4.2
  Zeichenkette, symbolische 4.3
Konventionen, typographische xiii
Koordinaten
  absolute
    STEP, angeben mit 5.9
    VIEW SCREEN, angeben mit 5.26
  Bildpunkte positionieren 5.5
  Grafiken und Text gegenübergestellt 5.6
  logische
    definiert mit WINDOW 5.29
    GET-Anweisung 5.60
    übertragen in physikalische Koordinaten 5.33
  physikalische
    Bildpunkte positionieren 5.29
    GET-Anweisung 5.60
    übertragen in logische Koordinaten 5.33
  relative
    Definition 5.9
    STEP 5.10
    VIEW 5.25
  Werte außerhalb des Darstellungsfeldes angeben 5.28, 5.62
Kreise, zeichnen 5.14
Kursiver Text, anzeigen von Platzhaltern xiii
KYBD 3.47

L

LBOUND-Funktion 2.20
LCASE$-Funktion 4.14
Leerzeichen, abschneiden
  linke Seite einer Zeichenkette 4.7, 4.8, 4.12
  rechte Seite einer Zeichenkette 4.7, 4.8, 4.12
LEFT$-Funktion 4.10
LEN-Funktion
  Beispielprogrammanwendung 3.40, 4.17
  Beschreibung 3.36
  Verwendung von 4.4, 4.5
Letzter Punkt, auf den Bezug genommen wurde 5.9, 5.10, 5.33
LINE INPUT#-Anweisung
  INPUT# im Unterschied zu 3.31
  INPUT$ im Unterschied zu 3.32
LINE INPUT-Anweisung im Unterschied zu INPUT 3.12
LINE-Anweisung
  Beispielanwendungen 5.73, 5.80, 5.86
  Reihenfolge von Koordinatenpaaren 5.9
  Schreibweise 5.8
  zeichnen
    ausgefüllte Rechtecke 5.12
    gepunktete Geraden 5.13
    nicht ausgefüllte Rechtecke 5.12
Literale Konstanten, Zeichenkette 4.2
LOC-Funktion
  Datenübertragung per Modem 3.65
  Geräte, mit 3.47
  SEEK im Unterschied zu 3.44
LOCATE-Anweisung
  Beispiel 3.51
  grafische Koordinaten im Unterschied zu 5.6
  positionieren der Ausgabe 3.4
  positionieren des Cursors 3.16
  verändern der Form des Cursors 3.17
LOF-Funktion
  Beispiel 3.56
  Dateien 3.40
  Geräte 3.48
Logische Koordinaten
  definieren mit WINDOW 5.29
  übertragen in physikalische Koordinaten 5.33
Logische Operatoren
  AND 1.4
  Boolesche Ausdrücke 1.4
  EQV 1.4
  IMP 1.4
  NOT 1.4
  OR 1.4
  XOR 1.4
Lokale Variablen 2.5, 2.7, 2.41
LPT1 3.47, 3.48
LPT2 3.47
LPT3 3.47
LSET-Anweisung 3.37
LTRIM$-Funktion
  abschneiden führender Leerzeichen 4.7, 4.12
  ausgeben von Zahlen ohne führende Leerzeichen 4.16
LTRIM$-Funktion (*Fortsetzung*)
  Zeichenketten fester und variabler Länge 4.8

M

Makrobefehl, DRAW 5.55
Mandelbrot-Menge 5.79
Marken xv
Mehrmodulige Programme
  Ereignisverfolgung 6.22
  Fehlerverfolgung 6.22, 6.23, 6.25
  FUNCTION-Prozeduren 2.7
  gemeinsame Benutzung von Variablen 2.37
  SUB-Prozeduren 2.4
MID$
  Anweisung 4.16
  Funktion 4.12, 4.16
MK*Typ*$-Anweisung 3.35
MK*Typ*MBF$-Anweisung 3.35
Modem, Datenübertragung mit 3.65
Module *Siehe* Mehrmodulige Programme
Monochromer Bildschirmmodus 5.34
Musik, Hintergrund 6.18
Musikereignisverfolgung
  ON PLAY GOSUB-Anweisung 6.18
  PLAY ON-Anweisung 6.19
Muster
  farbig 5.48, 5.52
  Formen ausfüllen 5.44, 5.73, 5.86
  Kachelgröße in 5.44
  monochrom 5.45
Musterkacheln, Editieren auf dem Bildschirm 5.86

N

Neue Zeile *Siehe* Wagenrücklauf-Zeilenvorschub-Folge
NOT-Operator
  Beschreibung 1.5
  logisch im Unterschied zu bitweise 1.5
  zusammengesetzte Prüfungen für Boolesche Ausdrücke 1.4

O

ON Ausdruck GOSUB-Anweisung 1.17
ON Ereignis GOSUB-Anweisung 6.27
ON ERROR GOTO-Anweisung
  erforderte Compileroption 6.27
  Schreibweise 6.3

ON PLAY GOSUB-Anweisung 6.19
OPEN COM-Anweisung 3.50, 3.64
OPEN-Anweisung
  Dateinamen in 3.22
  Dateinummern 3.22
  Dateizugriffsmodi
    APPEND 3.21, 3.27, 3.28
    BINARY 3.22, 3.43
    INPUT 3.21, 3.27
    OUTPUT 3.21, 3.26, 3.28
    RANDOM 3.22
  LEN-Klausel 3.36
Operatoren
  logische 1.4
  vergleichende 1.3, 4.7
OPTION BASE-Anweisung 2.10
OR-Operator 1.4, 5.62

P

PAINT-Anweisung
  *Siehe auch* Ausmalen
  Ausfüllen von Formen
    farbiges Inneres 5.42
    gemustertes Inneres 5.44, 5.73, 5.86
  Hintergrundargument 5.51
  Musterkachel
    farbig, Definition 5.48, 5.52
    monochrom, Definition 5.45
  numerischer Ausdruck 5.42
  Randargument 5.43, 5.50
  Zeichenkettenausdruck 5.44
PALETTE USING-Anweisung 5.39, 5.80
PALETTE-Anweisung 5.39
Paletten
  Bildschirmmodus 1 5.37
  COLOR, verändern mit 5.36
  PALETTE, verändern mit 5.39
Parameter
  Deklaration von Anzahl und Typ 2.23
  Format in DECLARE 2.23
  im Unterschied zu Argumenten 2.14
  Übereinstimmung mit Argumenten 2.15, 2.23
Pels *Siehe* Bildpunkte
Peripheriegeräte 3.47
Pfeiltasten *Siehe* RICHTUNGSTASTEN
Physikalische Koordinaten 5.33
PLAY ON-Anweisung 6.19
PLAY-Anweisung, Option für Hintergrundmusik 6.18
Plus-Operator (+), kombinieren von Zeichenketten 4.6
PMAP-Anweisung 5.80
PMAP-Funktion 5.33
POINT-Funktion 5.33
POS-Funktion 3.18, 3.51
Positive Zahlen, Ausgabe ohne führendes Leerzeichen 4.15
PRESET, Option der Grafikanweisung PUT Anweisung 5.62
PRESET-Anweisung
  Beschreibung 5.7
  verwenden der Farboption mit 5.7
PRINT USING-Anweisung 3.6, 3.51
PRINT#-Anweisung
  Begrenzung von Satzfeldern 3.30
  WRITE# im Unterschied zu 3.28
PRINT-Anweisung
  Ausgabe
    negative Zahlen 3.5
    positive Zahlen 3.5
  Beispiel 3.52
  Text umbrechen 3.5
  unterdrücken der Folge WR-ZV 3.5
  Zeichen
    Kommata 3.5
    Leerzeichen 3.5
    Semikolons 3.5
    Tabs 3.5
Programmausführung anhalten
  DO...LOOP-Anweisung 1.32
  FOR...NEXT-Anweisung 1.25
  TIMER-Anweisung 1.33
Programmierstil xv
Prozeduren
  Argumente übergeben
    als Referenz 2.7, 2.29
    als Wert 2.30
  aufrufen 2.11
  Ausdrücke 2.16
  automatische Variablen in 2.10, 2.40
  Datenfelder 2.15, 2.19
  definieren 2.9
  Format der Argumentenliste 2.15
  Format der Parameterliste 2.9, 2.15
  FUNCTION..END FUNCTION 2.9, 2.10
  in Quick-Bibliotheken 2.29
  Konstanten 2.16
  mehrmodulige Programme 2.24
  nicht erlaubte Anweisungen in 2.8
  rekursive 2.8, 2.42, 2.47
  STATIC-Variablen in 2.10, 2.40
  SUB...END SUB 2.9, 2.12
Prozeduren (*Fortsetzung*)
  Variablen
    globale 2.34
    lokale 2.5, 2.7, 2.41
  Verbunde 2.15, 2.21
  Vorteile 2.4
  Zusammenfassung 2.52
PSET-Anweisung
  Beispiel 5.80
  Beschreibung 5.7
  Farboption 5.7
  Option STEP 5.10
PSET, Option mit Grafikanweisung PUT 5.62
Punktierte Geraden, zeichnen 5.13
PUT-Anweisung
  Datei-E/A
    Binärzugriff 3.41, 3.43
    Direktzugriff 3.37, 3.38, 3.56
  grafische
    AND-Option 5.62
    Animation
      Option PSET 5.66
      Option XOR 5.65
    Bilder auf den Bildschirm kopieren 5.61
    Option OR 5.62
    Option PRESET 5.62
    Option PSET 5.62
    Option XOR 5.62
    Schreibweise 5.61
    steuern der Überlagerung mit dem Hintergrund 5.62

Q

Quadrate, zeichnen 5.22
Quick-Bibliotheken
  Fehlerverfolgung in 6.23
  lesen des Inhaltes von 3.45
  Prozeduren mit Include-Dateien deklarieren 2.29

R

Rechtecke
  ausgefüllt mit LINE 5.12
  gezeichnet mit LINE 5.11
Rechtecke, Festlegung von Koordinaten 5.12
REDIM-Anweisung, SHARED-Attribut 2.32, 2.34, 2.53
Registrierung 6.9
Rekursive Prozeduren
  anpassen der Stapelgröße für 2.44

Rekursive Prozeduren (*Fortsetzung*)
Definition 2.42
FUNCTION-Prozeduren 2.7
Relative Koordinaten *Siehe* Koordinaten, relative
RESTORE-Anweisung 6.6
RESUME NEXT-Anweisung
Beispiel 6.28
RESUME im Unterschied zu 6.7
RESUME-Anweisung
Beispiel 6.28
erforderte Compileroption 6.27
RESUME NEXT im Unterschied zu 6.7
RICHTUNGSTASTEN, verfolgen 6.13
RIGHT$-Funktion 4.12
Rollen, steuern mit VIEW PRINT 3.9
RSET-Anweisung 3.37
RTRIM$-Funktion 4.7, 4.8, 4.11
RUN-Anweisung, schließen von Datendateien 3.24

S

Sätze
Binärzugriffsdateien, schreiben 3.43
Definieren der Struktur von
FIELD 3.35
TYPE...END TYPE 3.35
Definition 3.19
Direktzugriffsdatei
hinzufügen zu 3.38
lesen 3.40
schreiben 3.39
wie gespeichert 3.34
fester Länge 3.21, 3.34
sequentielle Dateien
anfügen 3.20, 3.27
lesen 3.27
variabler Länge 3.25
wie gespeichert 3.24
übergeben an Prozeduren 2.15, 2.21
überschreiben existierender Daten 3.21, 3.26
Satznummern, Indizierung in einer Direktzugriffsdatei 3.41, 3.56
Schleifenstrukturen
Definition 1.19
DO...LOOP-Anweisung 1.27
FOR...NEXT-Anweisung 1.19
WHILE...WEND-Anweisung 1.25
Schlüsselwörter, Format in Programmbeispielen xv
Schriftbild
Befehlsnamen xiii
Schriftbild (*Fortsetzung*)
Beispiele xiii
Platzhalter xiii
Schlüsselwörter xiii
Tastennamen xiv
SCREEN-Anweisung
anpassen der Bildschirmauflösung mit 5.4
Auswirkung auf das Seitenverhältnis 5.23
Beispiel 5.73, 5.80
Bildschirmseite
aktive 5.71
sichtbare 5.51
Zeilenzahl im Textmodus 3.8
SCRN 3.47, 3.48
SEEK-Anweisung 3.44
SEEK-Funktion
Beschreibung 3.44
im Unterschied zu LOC 3.44
Seiten *Siehe* Bildschirmseiten
Seitenverhältnis
Berechnung für beliebige Bildschirmgröße 5.24
CIRCLE-Anweisung 5.15
Definition 5.23
Korrektur wegen 5.24
Quadrate zeichnen 5.22
SELECT CASE-Anweisung
CASE ELSE-Klausel 1.16
CASE-Klausel 1.13
END SELECT-Klausel 1.15
Fehlerbehandlungsroutine 6.4
IF...THEN...ELSE-Anweisung im Unterschied zu 1.11
ON Ausdruck GOSUB im Unterschied zu 1.17
Schreibweise 1.13
Semikolon
INPUT 3.5
PRINT 3.13
WR-ZV unterdrücken
Sequentielle Dateien
anfügen von Sätzen an 3.28
Direktzugriffsdateien, verglichen mit 3.20, 3.34
erstellen 3.20
Felder 3.26
hinzufügen von Sätzen zu 3.26
lesen von Sätzen aus 3.27
öffnen
APPEND 3.21, 3.28
INPUT 3.27
lesen 3.20
OUTPUT 3.21, 3.26
Sequentielle Dateien (*Fortsetzung*)
Sätze in 3.24
Serielle Datenübertragung
Definition 3.50
Eingabepuffer
EOF 3.50
LOC 3.50
LOF 3.50
Standardgröße 3.50
verändern 3.50
Serieller Kanal, öffnen für E/A 3.50
SHARED-Anweisung 2.32, 2.33
SHARED-Attribut
COMMON
Beschreibung 2.32
gemeinsam benutzen 2.33, 2.37
Zusammenfassung der Anweisungen 2.53
DIM
Beispiel 2.40
Beschreibung 2.32
gemeinsam benutzen 2.34
Verwendung, wann verboten 2.10
Zusammenfassung der Anweisungen 2.53
REDIM 2.32, 2.34, 2.53
Sinuskurve, grafisch darstellen 5.31
Sortieren, Beispiele 1.66, 1.34, 3.63
SPACE$-Funktion 4.14
Spalten, Verändern der Anzahl an, auf dem Bildschirm 3.8
SPC-Anweisung, wo sie erscheinen kann 3.8
Speichern von Bildern im Speicher mit GET 5.58
Standard-E/A
Definition 3.3, 3.11
in BASIC verwendete Anweisungen 3.67
Standardausgabe
Definition 3.3
umleiten 3.3
Standardeingabe
Definition 3.11
lesen
INKEY$ 3.14
INPUT 3.11
INPUT$ 3.14
LINE INPUT 3.12
Zeichen für Zeichen 3.13
Zeile für Zeile 3.12
umleiten 3.11
Stapelgröße, Anpassung für rekursive Prozeduren 2.44

STATIC
  Anweisung 2.41
  Attribut 2.10
  Variablen 2.40
STEP-Option, Grafikanweisungen akzeptieren 5.9
Stern (*), Zeichenkette fester Länge mit AS 4.3-4.5
Stop-Bits 3.50
STR$-Funktion 4.15
STRING$-Funktion 4.13
SUB-Anweisungen nicht erlaubt in Include-Dateien 2.29
SUB-Prozeduren
  Argumente ohne CALL 2.13
  CALL
    aufrufen mit 2.13
    aufrufen ohne 2.13
  Ereignisverfolgung in 6.21
  Fehlerverfolgung in 6.21
  GOSUB...RETURN im Unterschied zu 2.5
  lokale Variablen in 2.5, 2.41
SUB...END SUB-Prozeduren *Siehe* SUB-Prozeduren
Suchen
  binär 3.60
  Zeichenketten 4.8
Symbolische Konstanten
  Format in Programmbeispielen xv
  Zeichenkette 4.3
Syntaxanmerkungen
  Auswahlmöglichkeiten xiv
  optionale Größen xiii
  Platzhalter xiii

T

TAB-Anweisung
  Beispiel 3.51
  wo sie erscheinen kann 3.8
Tastatur, Lesen von Eingabe von 3.10
Tastenverfolgung
  aktivieren 6.13
  benutzerdefinierte Tasten
    einzelne Tasten 6.15
    umgeschaltete Tasten 6.15
  Funktionstasten 6.14
  Richtungstasten 6.14
Text-Darstellungsfeld *Siehe* Darstellungsfeld
Textausgabe
  Bildschirm 3.4, 3.48
  Drucker 3.48
TIMER-Anweisung, Programmausführung anhalten 1.33
Tortendiagramme, zeichnen 5.21
TYPE-Befehl (DOS) 3.28
TYPE...END TYPE-Anweisung
  Beispiel 3.57
  definieren von Sätzen in Direktzugriffsdateien 3.35
  im Unterschied zu FIELD 3.35
  Zeichenketten fester Länge in 4.5

U

Übergabe als Referenz
  Definition 2.29
  Prozeduren, verwendet in 2.7, 2.29
Übergabe als Wert
  DEF FN, verwendet in 2.7
  Definition 2.30
UBOUND-Funktion 2.20
UCASE$-Funktion 4.14
Umgeschaltete Tasten, verfolgen 6.15
Unbenannter Common-Block 2.46
Unterprogramm *Siehe* SUB-Prozeduren
Unterroutine *Siehe* GOSUB...RETURN-Unterroutine

V

VAL-Funktion 4.15, 4.17
Variablen
  *Siehe auch* Zeichenketten
  automatische 2.10, 2.40
  gemeinsam benutzen
    alle Prozeduren in einem Modul 2.32, 2.34
    mehrmodulige Programme 2.37
    Programme 2.44
    zwischen einer Prozedur und Modul-Ebene 2.32, 2.33
  globale
    adreßgleiche Variablen 2.39
    FUNCTION mit DEF, vergleichen 2.7
    gemeinsam benutzen 2.34
    SUB mit GOSUB, vergleichen 2.5
  lokale 2.5, 2.7, 2.41
  Namen, Format in Programmbeispielen xv
  STATIC 2.10, 2.40
  Typ deklarieren 2.18
  übergeben an Prozeduren
    als Referenz 2.7, 2.29
    als Wert 6.29, 6.30
  Zeichenkette 4.3
/v-Compileroption 6.27
Verfolgung
  *Siehe auch* Ereignisverfolgung, Fehlerverfolgung
  Ereignisse 6.9
  Fehler 6.2
  mehrmodulige Programme 6.22
  vom Compiler erforderte Befehlszeilenoptionen 6.27
Vergleichen von Zeichenketten
  fester und variabler Länge 4.8
  Vergleichsoperatoren, mit 4.7
Vergleichsoperatoren
  Boolesche Ausdrücke 1.3
  SELECT CASE-Anweisung 1.13
  Vergleiche von Zeichenketten 4.7
Verkettung 4.6
Verlassen
  einer Funktion, EXIT DEF 2.54
  Prozeduren
    EXIT FUNCTION 2.52
    EXIT SUB 2.52
VGA, Palette ändern 5.39
Video Grafikadapter *Siehe* VGA
VIEW PRINT-Anweisung 3.8
VIEW SCREEN-Anweisung 5.26
VIEW-Anweisung 5.25, 5.80, 5.86
Vordergrundfarbe
  Bildschirmmodus 1 5.37
  Standard 5.7
Vorwärtsbezug, Problem des
  Definition 2.22
  verwenden von DECLARE zur Vermeidung 2.22

W

Wagenrücklauf-Zeilenvorschub-Folge 1.10
Wahre Ausdrücke 1.4, 3.27
/w-Compileroption 6.27
WHILE...WEND-Anweisung
  DO...LOOP verglichen mit 1.27
  Schreibweise 1.25
WIDTH-Anweisung
  Spalten, verändern der Anzahl an 3.8
  Zeilen, verändern der Anzahl an 3.8
WINDOW SCREEN-Anweisung
  Auswirkung auf PUT 5.61
  WINDOW, verglichen mit 5.29
WINDOW-Anweisung
  Beispiel 5.80
  GET, Auswirkung auf 5.60
  Koordinaten 5.29
  PUT, Auswirkung auf 5.61

Winkel, von BASIC gemessen 5.17
Winkelmessungen
  Grad 5.17
  Radiant 5.18
WRITE#-Anweisung, im Unterschied zu PRINT# 3.28

X

/x-Compileroption 6.27
XOR, Operator 1.4
XOR, Option mit Grafikanweisung PUT 5.63

Z

Zahlen
  Ausgabe auf den Bildschirm 3.5, 4.15
  Darstellung als Zeichenketten 4.15
  positive, Ausgabe ohne führendes Leerzeichen 4.16
  Speicherung in Direktzugriffsdateien 3.34
Zeichen, wie sie gespeichert werden 4.4
Zeichenkette
  Ausdrücke
    *Siehe auch* Zeichenketten
    Begrenzung in sequentiellen Dateien 3.29
    Definition 4.3
  Variablen, Deklarierung *Siehe auch* Zeichenketten$ 4.3
    AS STRING-Klausel 4.3
    DEFSTR 4.3
  Verarbeitung *Siehe* Zeichenketten
Zeichenketten
  alphabetisieren 4.8
  Ausdrücke 4.3
  darstellen als Zahlen 4.15
  definiert 4.2
  ersetzen mit einer anderen Zeichenkette 4.16
  erzeugen 4.13
  fester Länge
    AS STRING 4.3
    variabler Länge im Unterschied zu 4.8
    Verbundelemente 4.5
  Konstanten
    literale 4.2
    symbolische 4.3
  Leerzeichen entfernen
    linke Seite 4.12
    rechte Seite 4.11
Zeichenketten (*Fortsetzung*)
  suchen nach einer Teilzeichenkette 4.9
  Variablen 4.3
  variabler Länge
    AS STRING 4.3
    definiert 4.4
    fester Länge verglichen mit 4.8
    maximale Größe 4.4
  vergleichen 1.3, 4.7
  Zeichen zurückbekommen von
    der linken Seite 4.10
    der rechten Seite 4.12
    Mitte 4.12
  Zusammenfassung der Anweisungen und Funktionen 4.18
  zusammensetzen 4.6
Zeichenketten fester Länge
  Parameterliste 2.15
  selbständige Variablen 4.5
  variable Länge verglichen mit 4.8
  Verbundelemente 4.5
Zeichenketten variabler Länge
  definiert 4.4
  GET-Anweisung 3.43
  maximale Größe 4.4
  PUT-Anweisung 3.43
  verglichen mit fester Länge 4.8
Zeilen, verändern der Anzahl an 3.8

Microsoft®

0188 Artikelnr. 01647

Microsoft®

Microsoft® QuickBASIC 4

Programmieren in BASIC: Ausgewählte Themen

0188 Artikelnr. 01647

# Microsoft® QuickBASIC 4

# Programmieren in BASIC: Ausgewählte Themen

Microsoft®